Einführung in JavaScript

Hier entsteht eine Dokumentation der Programmiersprache JavaScript. Sie richtet sich an Einsteiger, soll dem Leser aber nach und nach alle wichtigen Aspekte des JavaScript-Einsatzes bei der Webseiten-Entwicklung nahezubringen. Ziel ist ein profundes Verständnis der Sprache und ihre Beherrschung von den Grundlagen bis zur Entwicklung komplizierter Scripte und dem gezielten Einsatz von JavaScript-Frameworks.

Kommentare und Feedback werden gerne per E-Mail entgegen genommen.

JavaScript: Aufgaben und Anwendungsbereiche

Was ist unter JavaScript zu verstehen?

JavaScript ist eine Programmiersprache, die als Zusatztechnik in Webseiten eingebunden wird. Die JavaScript-Programme, auch

Scripte genannt, werden im Web-Browser interpretiert. Das heißt, sie werden in Maschinencode übersetzt und ausgeführt. Da JavaScripte auf dem Rechner des Websurfers ausgeführt werden, spricht man von einer *clientseitigen* Programmiersprache, um sie von Programmen abzugrenzen, die auf dem Web-Server laufen, wie etwa PHP- oder Perl-Scripte.

Was macht JavaScript und wie funktioniert JavaScript?

JavaScripte haben Zugriff auf das Browserfenster und das darin angezeigte HTML-Dokument. Ihre wichtigste Aufgabe besteht darin, auf Benutzereingaben im Dokument zu reagieren (z.B. klickt der Benutzer auf ein Element oder gibt einen Text in ein Formularfeld ein). JavaScripte können daraufhin Änderungen im gegenwärtig angezeigten HTML-Dokument vornehmen. Diese Änderungen finden nur im Browser, genauer gesagt im Arbeitsspeicher des Rechners statt, während das Dokument auf dem Web-Server unangetastet bleibt.

Die Änderungen können sowohl den Inhalt als auch die Darstellung des Dokuments betreffen. Auf diese Weise kann ein Dokument *interaktiv* und »*dynamisch*« gestaltet werden. Das bedeutet, es kann auf Benutzereingaben reagieren und sich ändern, ohne dass ein neues Dokument vom Web-Server abgerufen werden muss. Beispielsweise können Zusatzinformationen eingeblendet werden, sobald der Anwender mit der Maus auf ein bestimmtes HTML-Element klickt.

Den lesenden und verändernden Zugriff auf das Dokument regelt das sogenannte **Document Object Model** (DOM) – dazu später mehr. Der überragende Teil der JavaScript-Programmierung besteht aus dem Umgang mit dem DOM. Wegen dieser zentralen Wichtigkeit wird auch von »*DOM Scripting*« gesprochen. Ein älterer, mittlerweile überholter Begriff lautet »*Dynamisches HTML*«, abgekürzt DHTML. Dies soll Ihnen nur verdeutlichen, dass sich JavaScript in erster Linie darum dreht, mit dem HTML-Dokument zu hantieren, in dessen Kontext ein Script ausgeführt wird.

Bedeutung von JavaScript für das Web

JavaScript ist aus dem heutigen Web nicht mehr wegzudenken und nimmt neben HTML und CSS eine enorm wichtige Rolle ein. Beinahe jede Website verwendet kleinere oder größere JavaScripte.

Viele verbreitete Funktionen im Web lassen sich mit JavaScript vereinfachen und benutzerfreundlicher gestalten und neue Funktionen können hinzukommen. Das bedeutet allerdings nicht, dass Ihre Website JavaScript verwenden *muss*, um vollständig und erfolgreich zu sein. Viel JavaScript hilft nicht unbedingt viel. Ein bedachter JavaScript-Einsatz sorgt beispielsweise dafür, dass eine Website auch ohne JavaScript gut benutzbar und zugänglich ist.

Die Sprache JavaScript existiert schon seit über zehn Jahren und die Plattform wurde stetig fortentwickelt. Lange war JavaScript das »Schmuddelkind« unter den Webtechniken und seine Daseinsberechtigung wurde angezweifelt. Erst in den vergangenen Jahren kam es zu einer Renaissance von JavaScript und die JavaScript-Nutzung explodierte unter dem Schlagwort *Ajax*. Zudem entstanden Theorien, die den vernünftigen JavaScript-Einsatz begründen.

JavaScript als vollwertige Programmiersprache

Für viele Webautoren ist JavaScript die erste Programmiersprache, mit der sie zu tun bekommen. HTML und CSS sind zwar auch künstliche Rechnersprachen, aber keine *Programmiersprachen* – es lassen sich damit keine Programme schreiben, die Anweisung für Anweisung abgearbeitet werden.

Was zuerst wie eine spitzfindige Unterscheidung klingt, ist ein Unterschied ums Ganze: Mit HTML lassen sich lediglich *Texte auszeichnen* und Dokumente aufbauen. Mit CSS werden diese *Dokumente formatiert* und gelayoutet. In CSS werden sogenannte Deklarationen verfasst, die HTML-Elemente ansprechen und ihnen Formatierungen zuweisen.

Eine Programmiersprache hingegen funktioniert **grundlegend anders** und erfordert eine eigentümliche Denkweise und Herangehensweise an Probleme. Die Grundlagen der Programmierung können an dieser Stelle nicht beschrieben werden, sie sind aber essentielle Voraussetzung für den Umgang mit JavaScript.

Besonderheiten und Tücken von JavaScript

JavaScript ist eine Baustelle: Es hat sehr klein angefangen und wurde nicht entworfen, um den Anforderungen des heutigen Einsatzes zu genügen. Daher stecken der JavaScript-Kern und viele der über JavaScript nutzbaren Techniken voller Stärken und Schwächen – manches ist besonders einsichtig und einfach, anderes besonders verworren und macht Ihnen die Arbeit schwer.

Heutzutage beruhen große Webanwendungen auf umfangreichen und hochkomplexen JavaScripten. JavaScript gerät dadurch in

das Interesse von Informatikern und Software-Entwicklern, die die Sprache erforschen, mit ihr experimentieren und Grenzen austesten. Empfehlenswerte Programmiertechniken sind dadurch erst nach und nach ans Licht gebracht worden – und es ist ein andauernder Prozess, den Sie verfolgen sollten und an dem Sie sich beteiligen können.

JavaScript nimmt unter den Programmiersprachen einen besonderen Platz ein. Wenn Sie bereits andere Programmiersprachen kennen, werden Sie sicher Gemeinsamkeiten erkennen und Ihr Wissen wiederverwenden können. Doch früher oder später werden Sie auf Eigentümlichkeiten von JavaScript stoßen, die Sie so in keiner anderen im Web verbreiteten Programmiersprache finden werden. Sobald Sie komplexere Scripte schreiben, sollten Sie mit diesen Eigentümlichkeiten und den daraus resultierenden Programmiertechniken Bekanntschaft machen, denn Sie werden sie zu schätzen lernen.

JavaScript: Vorkenntnisse und Voraussetzungen

Herangehensweise dieser Einführung

Diese Dokumentation wählt eine *bestimmte* Herangehensweise an JavaScript und empfiehlt einen*bestimmten* JavaScript-Einsatz. Sie gibt Ihnen bewusst eine Arbeitsweise vor und geht von einigen Grundannahmen aus, die im Folgenden offen gelegt werden. Sie müssen mit diesen selbstverständlich nicht einverstanden sein und können Ihren eigenen Weg wählen.

- Diese Anleitung ist nicht für Webautoren gedacht, die bloß fremde Fertigscripte in ihre Sites einbauen wollen. Diese Einführung soll dazu befähigen, den Themenkomplex zu verstehen, um vom Einstieg an auf hohem Niveau zu programmieren. JavaScript wird nicht als schnell zu erlernende Technik dargestellt, sondern als eine vielseitige Programmiersprache mit erfreulichen Besonderheiten und nervenraubenden Tücken.
- Die Geschichte von JavaScript verlief äußerst wechselhaft. Viele der im Laufe der Geschichte entstandenen JavaScript-Tutorials sind veraltet oder empfehlen eine fragwürdige Praxis. Diese Anleitung beschränkt sich auf die aktuelle Situation und erklärt ausgewählte Grundlagen des gegenwärtigen JavaScript-Gebrauchs.
- Diese Einführung wirkt auf den ersten Blick trocken, weil sie derzeit aus viel erklärendem Text und wenig konkretem JavaScript-Code oder Anwendungsbeispielen besteht. Sie möchte zunächst die Grundkonzepte und später ständig benötigte Fertigkeiten vermitteln. Wenn Sie diese Grundlagen nachvollzogen haben, werden Sie den Aufbau von Scripten verstehen können und das Entwerfen eigener Scripte fällt Ihnen viel leichter.

HTML und CSS: Trennung von Struktur und Layout

Bevor Sie sich mit JavaScript beschäftigen, sollten Sie sich bereits intensiv mit HTML und CSS auseinandergesetzt haben. Idealerweise haben Sie bereits eine kleine Website mit diesen Web-Sprachen geschrieben. JavaScript tritt nämlich als *dritte, zusätzliche Technik* hinzu, die auf HTML und CSS aufbaut.

Für einen Einstieg in JavaScript ist es hilfreich, die *genauen Aufgaben* von HTML und CSS sowie vor allem deren Rolle im modernen Webdesign zu kennen:

- Sie sollten wissen, dass **HTML** der Auszeichnung von Texten dient. Damit entsteht ein strukturiertes Dokument – bestehend aus Überschriften, Abschnitten, Absätzen, Listen, Hyperlinks usw.
- Dieses Dokument wird anschließend mit **CSS** formatiert. Das Stylesheet hat dabei unter anderem die Aufgabe, die Struktur für den Leser ersichtlich zu machen, Übersichtlichkeit zu schaffen und nicht zuletzt eine prägnante, ansprechende und mitunter kunstvolle Erscheinung zu bieten.

Zentral beim zeitgemäßen Einsatz von HTML und CSS ist die **Trennung von Inhalt und Präsentation**. Das bedeutet, jegliche Informationen, die die Darstellung im Browser regeln, möglichst aus dem HTML-Code herauszuhalten und stattdessen ins Stylesheet auszulagern. Gleichzeitig werden inhaltliche und strukturelle Informationen (wie z.B. »dieser Text ist wichtig«) durch entsprechende Auszeichnungen im HTML-Code untergebracht, anstatt den Text bloß im Stylesheet z.B. fett zu formatieren.

Der Zweck dieser Trennung ist, dass der HTML-Code möglichst schlank ist und möglichst viel Bedeutung trägt. Man spricht daher auch von **semantischem Markup** (Semantik ist die Lehre von der Bedeutung). Dadurch lassen sich effiziente Stylesheets schreiben, die HTML-Elemente über gezielte Selektoren ansprechen. Im HTML-Code bedarf es dann nur wenige zusätzliche Angriffspunkte für das

Stylesheet wie etwa `div`-Abschnitte sowie `id`- und `class`-Attribute. Wenn ein zentrales Stylesheet das Layout zahlreicher Dokumente steuert, ist die Präsentation mit geringem Aufwand anpassbar, ohne dass alle betroffenen HTML-Dokumente geändert werden müssen.

Warum ist das für JavaScript wichtig? Je klarer und präziser HTML und CSS angewendet werden, desto einfacher ist es später, JavaScripte zu schreiben – denn mit JavaScript operieren Sie später auf Grundlage ihrer HTML-Dokumente und Ihre Scripte werden Hand in Hand mit dem Stylesheet zusammenarbeiten. Bevor Sie also die erste JavaScript-Codezeile schreiben, sollten Sie mit HTML und CSS beste Voraussetzungen dafür schaffen. Am Anfang mag das noch sehr theoretisch klingen, aber später werden Sie ein Gefühl dafür bekommen, wie HTML und CSS oganisiert sein müssen, um mit JavaScript effizient zusammenzuarbeiten.

Serverseitige Programmierung

Mit HTML und CSS alleine lassen sich nur statische Informationsangebote mit Texten, Bildern und Hyperlinks erstellen. Die meisten Websites bieten jedoch keine fertigen HTML-Dokumente, sondern interaktive Dienste. Websurfer durchsuchen Produktdatenbanken, vergleichen Preise, bestellen Artikel, schauen sich Foto-Diashows an, füllen Formulare aus und geben Texte ein, laden Dateien hoch dergleichen. Diese Funktionen basieren auf Programmen, die auf dem Webserver laufen. Die Programme fragen zumeist Datenbanken ab, verarbeiten die Browser-Anfrage und generieren HTML-Dokumente.

Der Einsatz von JavaScript lohnt sich insbesondere bei solchen interaktiven und personalisierten Webanwendungen, denn JavaScript hat die Möglichkeit, deren Bedienkomfort enorm zu verbessern. Das fängt bei einem Webforum an und geht bis zu solchen Anwendungen, die zum großen Teil aus JavaScript bestehen und sich dadurch wie gewöhnliche Desktop-Programme verhalten können.

Die vollen Fähigkeiten von JavaScript entfalten sich also im Zusammenhang mit Server-Anwendungen. Um JavaScript-Techniken wie Ajax einsetzen zu können, sollten Sie daher idealerweise Kenntnis einer serverseitigen Programmiersprache haben oder zumindest mit der Inbetriebnahme von serverseitigen Fertigscripten vertraut sein. Gängige Einsteigersprachen für serverseitige Dynamik sind PHP und Perl sowie Python und Ruby.

JavaScript: Grundlegende Konzepte

Clientseitige Scriptsprache

Bei JavaScript handelt es sich um eine Programmiersprache, die auf dem Client-Rechner direkt im Browser ausgeführt wird.

Um dies zu verstehen, müssen wir uns das Client-Server-Modell in Erinnerung rufen: Daten im Web werden mittels **HTTP** (englisch *Hypertext Transfer Protocol* = Hypertext-Übertragungsprotokoll) übertragen. Dieses Protokoll funktioniert nach einem Anfrage-Antwort-Schema. Der Rechner des Anwenders, genannt *Client* (englisch *Kunde*) stellt eine Verbindung zum *Server* (englisch *Diener*,*Anbieter*) her und schickt eine Anfrage. Dies kann z.B. die Anweisung sein, ihm eine bestimmte Adresse zu liefern. Der Webserver antwortet darauf in der Regel mit einer Bestätigung und sendet z.B. ein HTML-Dokument zurück.

JavaScript ist der Datenübertragung zwischen Anwender-Rechner und Webserver nachgeordnet. Scripte werden in ein HTML-Dokument eingebettet oder mit ihm verknüpft. Sie werden erst aktiv, wenn das HTML-Dokument beim Client-Rechner (zumindest teilweise) angekommen ist und der Web-Browser das Dokument samt Scripten verarbeitet.

Zusammenarbeit zwischen server- und clientseitigen Programmen

Ein häufiges Missverständnis ist, dass JavaScript mit serverseitigen Programmiersprachen gemischt werden kann. Die Ausführung von Serverprogrammen (wie z.B. PHP-Scripten) und die Ausführung von JavaScripten finden jedoch strikt *nacheinander und*

voneinander getrennt statt. Serverseitiger PHP-Code kann nicht direkt clientseitigen JavaScript-Code ausführen und umgekehrt, wie folgender Ablauf zeigen soll.

1. Der Client sendet eine HTTP-Anfrage an den Server, wenn der Anwender einen Hyperlink aktiviert oder ein Formular absendet.

2. Der Webserver nimmt die Anfrage entgegen. Das Serverprogramm wird gestartet und generiert üblicherweise ein HTML-Dokument, welches JavaScript enthalten kann. Dieser JavaScript-Code kann dynamisch durch das Serverprogramm zusammengesetzt werden.

   Das Serverprogramm läuft meistens nicht mehr als ein paar Sekunden und beendet sich dann. Damit endet der Wirkungsbereich des Serverprogramms: Es startet, um eine Client-Anfrage zu verarbeiten, und endet, bevor das generierte HTML-Dokument zum Client übertragen wird.

3. Der Browser auf dem Client empfängt den HTML-Code und verarbeitet ihn. In diesem Moment werden mit dem Dokument verknüpfte JavaScripte ausführt.

Dies soll zeigen, wie server- und clientseitige Programme zusammenarbeiten, aber doch getrennt sind – denn sie laufen nacheinander auf verschiedenen Rechnern, operieren in einer unterschiedlichen Umgebung und erfüllen andere Aufgaben.

Zu allem Überfluss ist JavaScript nach der Darstellung des HTML-Dokuments im Browser nicht auf dem Client eingeschlossen, sondern kann von sich aus mit Serverprogrammen interagieren. JavaScripte können allerdings nicht direkt Programmcode auf dem Server aufrufen. Die einzige Möglichkeit ist, eine HTTP-Anfrage im Hintergrund zu starten. Diese Technik werden wir später unter dem Begriff **Ajax**kennenlernen.

Dokument-Objektmodell (DOM), Elementobjekte und Knoten

Nachdem wir die Übertragung von HTML-Dokumenten vom Webserver zum Browser betrachtet haben, müssen wir uns vor Augen führen, wie der Browser das Dokument verarbeitet.

Der HTML-Code liegt dem Browser zunächst als bloßer Text vor. Noch während der Browser den Code über das Netz empfängt, verarbeitet er ihn Stück für Stück. Diese Aufgabe übernimmt der sogenannte *Parser* (englisch *parse* = einen Satz in seine grammatikalischen Einzelteile zerlegen). Der Parser überführt den HTML-Code in eine Objektstruktur, die dann im Arbeitsspeicher vorgehalten wird. – Mit *Objekt* ist hier ein Bündel von Informationen im Speicher gemeint. – Diese Objektstruktur besteht aus verschachtelten **Knoten**, allen voran Elementknoten, Attributknoten und Textknoten, die in einer Baumstruktur angeordnet sind.

Der Browser nutzt für alle weiteren Operationen diese Objektstruktur, nicht den HTML-Quellcode, an dem der Webautor üblicherweise arbeitet. Insbesondere CSS und JavaScript beziehen sich nicht auf den HTML-Code als Text, sondern auf den entsprechenden Elementenbaum im Speicher. Wie diese Objektstruktur aufgebaut ist, ist im besagten **Document Object Model** geregelt.

JavaScript-Interpreter

Wie kommen nun JavaScripte ins Spiel? JavaScript-Code wird direkt in HTML-Dokumente eingebettet oder indirekt mit externen Dateien eingebunden. Der Browser ruft den JavaScript-Code ebenfalls vom Web-Server ab und führt ihn aus, noch während das Dokument Stück für Stück analysiert (geparst) und dargestellt wird. Dazu verfügen die heutigen Web-Browser über einen eingebauten**JavaScript-Interpreter**.

Doch nicht alle Browser verfügen über einen solchen, schon gar nicht alle Programme, die in irgendeiner Weise Webseiten verarbeiten. Zudem verfügen die jeweiligen Interpreter über sehr unterschiedliche Fähigkeiten. Nicht zuletzt gibt es viele Gründe, warum ein JavaScript trotz vorhandenem Interpreter nicht ausgeführt wird oder nicht alle JavaScript-Techniken nutzen kann: Der Anwender kann den Interpreter für alle Websites oder für eine bestimmte abschalten. Die Ausführung von JavaScripten kann aus Sicherheitsgründen gesperrt oder beschränkt sein. Oder ein sogenannter Proxy, ein vermittelnder Rechner zwischen Client und Server, kann den Code während der Übertragung herausfiltern.

Objektbasierung

JavaScript basiert auf sogenannten Objekten, die die Umgebung repräsentieren, in der ein Script arbeitet. Der Zugriff auf das Browserfenster sowie das Dokument erfolgt über Objekte, die nach ganz bestimmten Regeln (Objekt-Typen) aufgebaut sind.

JavaScript-Objekte sind allgemein gesprochen Container für weitere Informationen. Ein Objekt funktioniert wie eine Zuordnungsliste, die unter einem Namen einen bestimmten Wert speichert. Diese Einträge werden *Eigenschaften*, im Englischen *Properties* oder *Member* genannt. Objekte können auf diese Weise beliebig verschachtelt werden, sodass ein Objekt Unterobjekte als Eigenschaften enthält.

Wie gesagt besteht das HTML-Dokument aus Sicht des DOM aus Knoten: Diese Knoten werden im JavaScript ebenfalls als Objekte angesprochen. Wir haben es also mit Knotenobjekten zu tun, die häufigsten sind Elementobjekte.

Eines sei vorweggenommen: Wenn wir eine Eigenschaft bzw. ein Unterobjekt eines Objektes ansprechen wollen, so notieren wir im JavaScript-Code den Namen des einen Objektes, dann einen Punkt und schließlich den Namen der Eigenschaft. Beispielsweise: `objekt.eigenschaft`.

Fenster und Dokumente

JavaScripte werden immer im Kontext eines HTML-Dokuments ausgeführt, das in Form eines Objektes namens `document` vorliegt. Zu jedem solchen Dokumentobjekt gehört immer ein Fensterobjekt. Es trägt den Namen **`window`** und wird **globales Objekt** genannt, weil es in der Objekthierarchie von JavaScript das oberste und wichtigste ist. Daran hängt das Dokumentobjekt`document` als Unterobjekt.

Mit *Fenster* ist in der Regel ein gewöhnliches Browserfenster gemeint – lediglich im Falle von Frames, Inner Frames und Popup-Fenstern werden in einem Browserfenster mehrere Dokumente mit mehreren Fensterobjekten dargestellt. Dass Sie mit JavaScript Zugriff auf das Fensterobjekt haben, bedeutet nicht, dass Sie den gesamten Browser unter Ihrer Kontrolle haben. Das Fensterobjekt ist bloß eine Repräsentation und dient als ordnendes Objekt, um darin gewisse Funktionen und Eigenschaften unterzubringen.

Ein JavaScript wird immer im Kontext genau eines Dokuments und genau eines Fensterobjekts ausgeführt. Wenn mehrere zusammenhängende Dokumente parallel im Browser angezeigt werden, wie etwa bei Frames, kann ein JavaScript jedoch auch auf andere Fensterobjekte und Dokumente zugreifen – zumindest prinzipiell, denn es gibt bestimmte Sicherheitshürden.

Dies sind nur die Grundlagen der beiden zentralen JavaScript-Konzepte. Genaueres erfahren Sie im Kapitel über Fenster und Dokumente sowie im Kapital über Sicherheit.

Ereignisse (Events)

JavaScripte arbeiten nicht im leeren Raum, sondern wie gesagt im Rahmen eines Browserfensters, in dem ein HTML-Dokument dargestellt wird. Wie erfolgt nun die Verbindung von HTML und JavaScript? Wann wird ein Script aktiv?

In dem Moment, in dem der Browser den HTML-Code einliest, in die besagte Objektstruktur überführt und die Darstellung berechnet, werden auch alle mit dem Dokument verknüpften Scripte ausgeführt.

Ihre Hauptarbeit leisten Scripte aber nicht in diesem Moment des Ladens der Seite. Sie werden zwar kurz aktiv, legen sich dann üblicherweise wieder »schlafen«. Sie werden erst wieder aktiv, wenn etwas im Dokument passiert. Denn JavaScript arbeitet nach dem Prinzip der **Überwachung und Behandlung von Ereignissen** (im Englischen *event handling*). Ein »*Ereignis*« kann vieles sein. Es geht dabei vor allem um Benutzereingaben. Beispielsweise:

- Auf der Tastatur wird eine Taste gedrückt (Ereignis-Typ: `keypress`)
- Ein Element im Dokument wird mit der Maus angeklickt (`click`)

Neben diesen Ereignissen, die direkt auf Benutzereingaben wie Tastendrücke und Mausklicks zurückgeben, gibt es unter anderem solche, die sich auf die Interaktion mit Formularen beziehen:

- Ein Formularfeld wird fokussiert (`focus`)
- Der Text in einem Formularfeld wird geändert (`change`)
- Ein Formular wird abgesendet (`submit`)

Schließlich können auch Ereignisse überwacht werden, die sich auf das ganze Dokument oder das zugehörige Fenster beziehen:

- Das Dokument wurde vollständig mitsamt aller Grafiken und Stylesheets geladen, seine Objektstruktur steht vollständig zur Verfügung (`load`)

- Die Größe des Browserfensters wird verändert (`resize`)

Ein solches Ereignis lässt sich nun mit der Ausführung einer JavaScript-Funktion verbinden (Teilprogramme werden in JavaScript in sogenannten *Funktionen* gruppiert - dazu später mehr im entsprechenden Kapitel). Immer wenn das Ereignis eintritt, wird dann die angegebene Funktion ausgeführt. Auf diese Weise kann auf eine Aktion des Anwenders eine JavaScript-Reaktion folgen.

Um ein Ereignis zu überwachen, sind drei Bestandteile nötig:

1. Die Stelle, **wo** das Ereignis überwacht werden soll: Fensterweit, dokumentweit oder nur an einem bestimmten Element im Dokument.
2. Der **Typ** des Ereignisses. Zum Beispiel `click`, das bedeutet alle Mausklicks.
3. Die JavaScript-**Funktion**, die beim Ereignis ausgeführt werden soll. Diese Funktion wird auch**Handler** oder **Handler-Funktion** genannt (englisch `handle` = verarbeiten, abwickeln).

Mit JavaScript können wir nun solche Regeln formulieren: *Überwache das Ereignis `click` beim Element mit der ID `button` und führe die Handler-Funktion `begrüßung` aus, wenn das Ereignis eintritt.*Klickt der Benutzer auf das besagte Element, wird die Funktion `begrüßung` ausgeführt und darin kann auf die Benutzereingabe reagiert werden.

Mit Fug und Recht kann behauptet werden, dass das *event handling* das folgenreichste Konzept von JavaScript ist: Die heutige Programmierung von JavaScripten besteht zum allergrößten Teil aus dem Entwerfen Ereignis-gesteuerter Abläufe. Gleichzeitig bringt dieser Bereich große Herausforderungen mit sich, die die Praxis ungemein erschweren.

Genaueres zum Thema können Sie im Kapitel Grundlagen zur Ereignisverarbeitung erfahren.

Sinnvoller JavaScript-Einsatz

Welche Aufgabe nimmt JavaScript im Webdesign ein?

Die Aufgabe und der Zweck von JavaScript ist nicht ein für alle Mal festgelegt, sondern hat sich im Laufe der Zeit immer wieder gewandelt. Dazu tragen Dokumentationen, Fachartikel und Scripte bei, die ein bestimmtes Verständnis von JavaScript verbreiten. Die Verwendung von JavaScript ist nie eindeutig positiv oder eindeutig negativ zu sehen. Heute gibt es keine einheitliche Auffassung davon, wozu JavaScript überhaupt gut sein soll und wann es besser vermieden werden sollte.

JavaScript blickt auf eine düstere Vergangenheit zurück, in der die Sprache vor allem für unnütze Spielereien, bedeutungslose »Dynamik« oder sogar zur Gängelung der Anwender missbraucht wurde. Anstatt Inhalte einfacher zugänglich zu machen, erschwerten oder verhinderten manche Scripte den Zugang. Dadurch haftete JavaScript lange Zeit ein zweifelhafter Ruf an.

Bis vor einigen Jahren verlief der JavaScript-Gebrauch weitgehend ungelenkt und es fehlte ein theoretischer Unterbau, der die sinnvolle Anwendung von JavaScript begründete. Es gab bereits einfache Faustregeln darüber, wann und wie JavaScript eingesetzt werden sollte: JavaScript sei nur ein optionaler, das heißt weglassbarer Zusatz, der die Bedienung vereinfachen und die Benutzerfreundlichkeit steigern soll. Daran hielten sich allerdings nur wenige Webautoren. Auch fehlte eine allgemeine Theorie, die die Rolle von JavaScript im Webdesign bestimmte.

Fest steht, dass die gezielte Aufwertung von Webseiten mit JavaScript die Bedienung maßgeblich verbessern *kann*. In diesem Abschnitt sollen die Voraussetzungen dafür untersucht werden. Die vorgestellten Theorien sind nicht der Weisheit letzter Schluss,

sondern lediglich der aktuelle Stand einer Debatte, die sich stets weiterentwickelt.

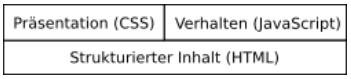
Das Schichtenmodell und die Verhaltens-Schicht

Im Zuge der sogenannten Webstandards-Bewegung setzen sich viele Webentwickler für einen sinnvollen und korrekten Einsatz der Webtechniken HTML und CSS ein. Das Ziel waren inhaltsreiche, barrierefreie und anpassungsfähige Websites. In diesem Zusammenhang wurden die Grundlagen für modernes JavaScript erarbeitet, wie etwa die Trennung von Struktur und Layout.

Während der HTML- und CSS-Gebrauch zu dieser Zeit revolutioniert wurde, blieb JavaScript lange Zeit ausgeklammert. Erst ab dem Jahre 2004 machten sich einige aus der Webstandards-Bewegung daran, auch JavaScript einen neuen Sinn zu verleihen und den »richtigen« JavaScript-Gebrauch zu erforschen. Heraus kam das sogenannte **Schichtenmodell**, das den drei Webtechniken HTML, CSS und JavaScript gewisse Funktionen zuweist und sie aufeinander aufbauen lässt.

Wir haben bereits in der Einleitung besprochen, dass modernes Webdesign den **strukturierten Inhalt** (Text mit HTML-Auszeichnungen) von der **Präsentationslogik** trennt: Die Informationen zur Präsentation werden aus dem Markup in ein zentrales CSS-Stylesheet ausgelagert. Das Stylesheet baut auf dem Markup auf und ergänzt es – das HTML-Dokument soll aber auch ohne das Stylesheet möglichst zugänglich sein.

So ensteht das Schichtenmodell: HTML bietet die grundlegende Schicht, darüber liegt die CSS-Schicht. Im Code sind beide Schichten voneinander getrennt, um optimale Wartbarkeit, Ausbaubarkeit und Flexibilität zu gewährleisten. Dieses Modell lässt sich zunächst in einem einfachen Diagramm veranschaulichen:

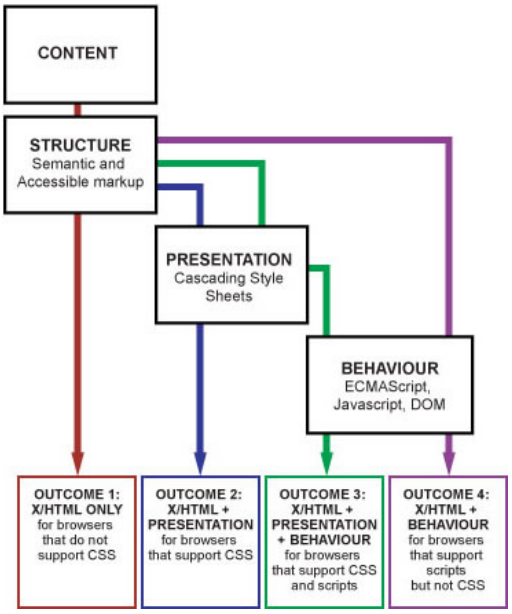


Der Clou ist nun, JavaScript ebenfalls als eine solche Schicht zu begreifen, genannt Behaviour Layer, zu deutsch **Verhaltens-Schicht**.

Mit »Verhalten« ist Interaktivität gemeint: *Wenn der Anwender etwas tut, bringt JavaScript die Webseite dazu, sich in einer bestimmten Weise zu verhalten.* Wie CSS fügt JavaScript dem Dokument einen besonderen Mehrwert hinzu. CSS und JavaScript sollen mit dem Ziel eingesetzt werden, die Benutzbarkeit zu verbessern.

Die drei Schichten HTML, CSS und JavaScript arbeiten Hand in Hand, aber außer der HTML-Schicht ist keine für das grundlegende Funktionieren notwendig. Insbesondere funktioniert die Präsentation auch dann, wenn JavaScript nicht zur Verfügung steht. Das »Verhalten« funktioniert seinerseits soweit wie möglich, wenn das Stylesheet nicht angewendet wird. Damit hat man z.B. sehbehinderte Anwender mit Screenreadern im Kopf.

Wenn HTML die Basis bildet und die Zusätze CSS und JavaScript wegfallen können, gibt es vier verschiedene Kombinationsmöglichkeiten. Das folgende Schaubild schildert die die vier Fälle, die berücksichtigt werden sollten:



(Die Grafik wurde übernommen aus dem Webstandards-Workshop von Russ Weakley.)

TODO: Übersetzung

Schrittweise Verbesserung und Abwärtskompatibilität

Dieses Schichtenmodell definiert JavaScript als optionale Erweiterung – mit allen Konsequenzen für das Webdesign. Zwar ist der Fall am häufigsten, dass sowohl das Stylesheet als auch JavaScripte umgesetzt werden. Aber die Website muss immer noch funktionieren, wenn der Browser diese nicht, nur unvollständig oder gar eigenwillig umsetzt.

Dahinter steckt ein allgemeines technisches Modell namens **Progressive Enhancement** (zu deutsch in etwa: *schrittweise Verbesserung*). Man startet auf einer einfachen, grundlegenden Ebene, in diesem Fall HTML. Schritt für Schritt fügt man weitere Bausteine hinzu, die die Attraktivität der Webseite erhöhen: Stylesheets, Grafiken, Scripte, eventuell Animationen, Flash-Filme, Java-Applets, Multimedia-Dateien usw. Entscheidend ist, dass beliebig hoch gebaut werden kann, aber die minimalen Eingangsvoraussetzungen gleich bleiben.

Unterstützt ein Client-Rechner also nur bestimmte Teiltechniken, bekommt der Anwender soviel wie möglich zu sehen. Damit wird garantiert, dass jeder Anwender die Website zumindest rudimentär nutzen kann und der wichtige Inhalt für automatisierte Programmme wie z.B. Suchmaschinen-Crawler zugänglich bleibt.

Unaufdringliches JavaScript (Unobtrusive JavaScript)

Eine Weiterführung des Schichtenmodells stellt das Konzept **Unobtrusive JavaScript** dar. Dabei handelt es sich um einen Regelkatalog für eine vorteilhafte JavaScript-Anwendung. »Unobtrusive« bedeutet unaufdringlich, unauffällig und dezent. Das Wort wird in diesem Zusammenhang aber auch mit »barrierefrei« übersetzt.

Die Kernidee hinter unaufdringlichem JavaScript ist Zurückhaltung und Vorsicht. Viele der Regeln haben wir bereits kennengelernt:

- JavaScript ist eine optionale Erweiterung, auf die man sich nicht verlassen sollte.
- HTML-, CSS- und JavaScript-Code sollten nicht vermischt werden.
- Das Script sollte möglichst robust sein. Es fragt die Verfügbarkeit der Objekte ab, die es nutzen will. Das Script geht souverän mit den Fällen um, in denen etwas Ungewöhnliches passiert. Es treten somit keine kritischen Fehler auf.
- Das Script orientiert sich möglichst an offenen, breit unterstützten Webstandards, anstatt Lösungen zu verwenden, die auf alten, herstellereigenen Techniken basieren und nur in bestimmten Browsern funktionieren.
- Da unbekannt ist, mit welchen Eingabegeräten der Anwender die Webseite bedient, nutzt das Script geräteunabhängige JavaScript-Ereignisse oder berücksichtigt Maus- und Tastatureingaben gleichermaßen.
- Das Script ist in sich abgeschlossen, sodass es problemlos mit anderen Scripten zusammenarbeiten kann, ohne dass die Scripte sich in die Quere kommen.
- Das Script wird so übersichtlich strukturiert, dass es auch für Fremde verständlich, einfach anpassbar und damit wiederverwendbar ist. Das Script arbeitet rationalisiert und möglichst effizient.

Wie sich diese Regeln in der Praxis umsetzen lassen, werden wir später noch erfahren.

Besonderheiten bei Webanwendungen (Ajax)

Das einfache Credo, JavaScript nur als nützliche, aber weglassbare Zusatzschicht zu verwenden, hatte solange Bestand, wie JavaScript zu nichts anderem fähig war, als »statische« HTML-Dokumente mit ein wenig Interaktivität aufzumotzen. Dies hat sich in den letzten Jahren schlagartig geändert mit dem Aufkommen von sogenannten Webanwendungen, bei denen JavaScript die Hauptrolle spielt und die ohne JavaScript überhaupt nicht möglich wären. Das Zauberwort, dass diese JavaScript-Offensive einleitete, lautet **Ajax**.

Was unter Ajax zu verstehen ist und wie die neuen Webanwendungen aufgebaut sind, nimmt das Kapitel Serverkommunikation und dynamische Webanwendungen genauer unter die Lupe. An dieser Stelle sei nur angemerkt, dass JavaScript dabei nicht mehr bloß auf Basis klassischer HTML-Dokumente mit abwärtskompatiblen Textinhalten operiert. Stattdessen stellt die Script-Logik die gesamte Funktionalität zur Verfügung.

Das bedeutet: Nimmt man die Scripte weg, bleibt ein unbrauchbares Gerüst übrig, und es wird selten eine Alternative angeboten.

Aus Sicht der genannten Auffasungen vom sinnvollen JavaScript-Einsatz ist das der absolute Albtraum. Und doch genießen solche hochdynamischen Webanwendungen eine ungeheure Popularität und vereinfachen die Arbeit im Netz, sodass niemand die Berechtigung dieses JavaScript-Einsatzes bezweifelt.

Dennoch darf man aus dem Ajax-Boom nicht die Konsequenz ziehen, die bisherigen Richtlinien über Bord zu werfen und JavaScript als selbstverständliche Zugangsvoraussetzung für alle möglichen Websites erklären. Klassische Websites und auch stark interaktive Webanwendungen können mit »unaufdringlichem« JavaScript nennenswert verbessert werden, ohne dass Zugänglichkeit und Kompatibilität dabei zu kurz kommen. Gleichzeitig macht es in manchen Fällen Sinn, JavaScript und gewisse Browser-Fähigkeiten zwingend vorauszusetzen. Aber auch Webanwendungen, die ohne JavaScript nicht auskommen, können eine abwärtskompatible Alternativ-Version bereitstellen, um die Zugänglichkeit zu gewährleisten.

Zugänglichkeit und Barrierefreiheit

Anders als landläufig vermutet wird, nutzen gerade Menschen mit Behinderungen verstärkt das World Wide Web. Barrierefreiheit soll gewährleisten, dass Behinderte eine Webseite möglichst problemlos lesen und bedienen können. Die Barrierefreiheit berücksichtigt sowohl die körperlichen und geistigen Besonderheiten der Webnutzer als auch die Anforderungen der verwendeten Web-Zugangstechniken. Ziel ist es, dass die *Zugänglichkeit* der Webseite in möglichst allen Fällen gewährleistet ist.

Ein prägnantes Beispiel ist der stark sehbehinderte oder sogar blinde Web-Nutzer, der eine Webseite mithilfe eines sogenannten Screenreaders liest. Ein *Screenreader* ist ein Zusatzprogramm, das Web-Inhalte mittels Sprachausgabe vorliest oder sie über eine sogenannte Braille-Zeile ertastbar macht. Ein weniger extremes Beispiel ist ein älterer Mensch, der von der Unübersichtlichkeit und der kleinen Schrift einer Webseite überfordert ist. Gemäß einer erweiterten Definition ist es auch eine Frage der Zugänglichkeit, ob eine Website über ein Mobiltelefon bedient werden kann und ob ein sogenannter Web-Crawler – ein automatisches Indizierungsprogramm einer Web-Suchmaschine – zu den Inhalten vordringen kann.

Der maßgebliche Standard, der Kriterien und Prüfverfahren für eine barrierefreie Website festlegt, heißt *Web Content Accessibility Guidelines* (engl. **Zugänglichkeits-Richtlinien für Web-Inhalte**, abgekürzt **WCAG**). Die Richtlinien liegen in Version 2 vor und werden vom World-Wide-Web-Konsortium (W3C) herausgegeben.

Bei Webauftritten der öffentlichen Verwaltung ist Barrierefreiheit in vielen Ländern gesetzliche Pflicht, aber auch Organisationen und Unternehmen sehen die Barrierefreiheit zunehmend als Erfordernis. In Deutschland regelt die *Barrierefreie Informationstechnik-Verordnung* (BITV) auf Bundes- und Länderebene die Anforderungen an öffentliche Websites.

Der Effekt von JavaScript auf die Zugänglichkeit ist nicht einfach positiv oder negativ zu bewerten. Festzuhalten ist, dass JavaScript schwerwiegende Zugangsbarrieren aufbauen kann. Leider gibt es keine einfachen Regeln, um Barrieren bei den üblichen JavaScript-Anwendungen zu vermeiden, und die Auswirkung des exzessiven JavaScript-Gebrauchs ist nur in Ansätzen erforscht.

In der JavaScript-Entwicklung haben wir meist folgende Situation vor Auge: Ein Anwender sitzt vor einem leistungsstarken Desktop-Rechner mit hochentwickeltem Browser und exzellenten JavaScript-Fähigkeiten. Er nimmt die Webseite über einen großen, hochauflösenden Bildschirm wahr und bedient den Rechner gekonnt mit Tastatur und einer Maus. Er findet sich auf einer Website sofort zurecht, kann schnell in ihr navigieren und Text, grafische und Multimedia-Inhalte wahrnehmen. – Dies ist aus Sicht der Barrierefreiheit bloß ein Idealfall. Faktisch nutzen Menschen zahlreiche Zugangstechniken, um Webseiten zu betrachten, und haben individuelle Fähigkeiten und Bedürfnisse. Der Barrierefreiheit geht es darum, Webseiten so zu entwickeln, dass sie dieser Vielfalt gerecht werden.

JavaScript lebt davon, auf Benutzereingaben zu reagieren, Änderungen an Inhalt und Darstellung des Dokuments vorzunehmen und somit in Interaktion mit dem Anwender zu treten. All diese Schritte der JavaScript-Programmierung stehen in Frage, wenn man alternative Zugangstechniken und Benutzereigenheiten berücksichtigt: Beispielsweise ein Screenreader wird oft alleine mit der Tastatur bedient, Änderungen am Dokument bekommt ein Screenreader-Nutzer nicht unbedingt mit und die Interaktion müsste völlig anders ablaufen, um ihn zu erreichen.

Diese Problematik soll hier nur angerissen werden. ... TODO

Empfehlungen und Leitlinien für die Praxis

Wir haben mehrere Modelle betrachtet, die uns bei der Entscheidung helfen können, wann und wie JavaScript sinnvoll einzusetzen ist. Webanwendungen haben wir als berechtigte Ausnahmen wahrgenommen, die die allgemeine Regel eher bestätigen. Sie können sich von diesen Grundregeln leiten lassen:

- Entwerfen Sie zunächst eine HTML-Struktur, in der alle wichtigen Inhalte auch ohne JavaScript zugänglich sind. Reichern Sie das Dokument mit passender Textauszeichnung an (Überschriften, Textabsätze, Listen, Hervorhebungen, `div`-Abschnitten usw.).
- Arbeiten Sie nach dem Schichtenmodell. Beachten Sie die eigentlichen Aufgaben und Zuständigkeiten von HTML, CSS und JavaScript und vermeiden Sie es, deren Code zu vermischen. Sorgen Sie für eine Trennung der drei Schichten bei gleichzeitiger optimaler Zusammenarbeit. Reizen Sie die Fähigkeiten der einen Technik aus, um dann mit der anderen Technik nahtlos daran anzuknüpfen.
- Schreiben Sie »unaufdringliche« Scripte, die sich automatisch dazuschalten und dem Dokument zusätzliche Interaktivität hinzufügen, sofern JavaScript verfügbar ist.
- Sie sollten stets die unterschiedlichen Umgebungen im Auge haben, unter denen ihre Scripte arbeiten müssen. Versuchen Sie, die Scripte tolerant gegenüber verschiedenen Zugangssystemen zu entwickeln. Setzen Sie keine Techniken voraus, sondern prüfen Sie deren Verfügbarkeit, nutzen Sie im Fehlerfall Alternativen oder beenden Sie das Script zumindest geordnet. Ihre JavaScripte sollten mit praktischen Schwierigkeiten souverän umgehen können.
- Sorgen Sie für aufgeräumten, gut strukturierten und verständlichen Code. Lernen Sie, wie Sie schnelle, effiziente und kompatible Scripte schreiben können.

Was das alles konkret bedeutet, mag Ihnen noch schleierhaft erscheinen - das ist nicht schlimm, die Ratschläge sollen bloß eine grobe Orientierung bieten. Sie werden mit zunehmender Erfahrung lernen, mit welchen Methoden diese Anforderungen zu bewerkstelligen sind.

JavaScript: Syntax-Grundlagen

Notizen

siehe Weblog-Artikel: Geheimnisse der JavaScript-Syntax

Aufbau eines JavaScript-Programmes

Ein JavaScript-Programm ist nach gewissen wiederkehrenden Regeln aufgebaut. Solche Syntax-Regeln kennen Sie aus der natürlichen, zum Beispiel der deutschen Sprache. Wenn Ihr Gesprächspartner Sie verstehen soll, müssen Sie beim Formulieren eines Satzes gewisse Grundregeln der Syntax und Grammatik einhalten, damit Sie verstanden werden.

Glücklicherweise verstehen unsere Mitmenschen uns auch dann, wenn wir kleinere Fehler machen und uns nur grob an die Regeln halten. Programmiersprachen sind jedoch viel strenger: Zum einen sind ihre Regeln vergleichsweise einfach, eindeutig und lassen wenig Spielraum. Zum anderen müssen sich Programmierer an diese Regeln halten und dürfen keine Ausnahmen machen.

Dies hat folgende Bewandnis: Damit der JavaScript-Interpreter ein Programm ausführen kann, muss er zunächst dessen Syntax

verstehen. Dieses Aufsplitten in die Bestandteile nennt sich **Parsing**. Wenn der Interpreter dabei auf einen Syntaxfehler trifft, bricht er mit einer Fehlermeldung ab und das JavaScript-Programm wird gar nicht erst ausgeführt.

Die folgende Beschreibung der JavaScript-Syntax ist sehr theoretisch und formal. Sie soll ihnen den groben Aufbau eines JavaScript-Programmes vermitteln, damit Sie wissen, welche Bestandteile an welchen Stellen vorkommen dürfen.

Anweisungen (Statements)

Vereinfachend gesagt besteht ein JavaScript-Programm aus einer Abfolge von einer oder mehreren Anweisungen, sogenannten Statements. Bei der Ausführung des JavaScripts werden die Anweisungen nacheinander abgearbeitet.

(Der Einfachheit halber rechnen wir die Funktionsdeklaration zu den Anweisungen, was der ECMAScript-Standard nicht tut. Aber Sonderfall der Funktionsdeklarationen, sie werden nicht in der Reihenfolge der anderen Anweisungen ausgeführt. Dasselbe bei Variablen-Deklarationen. TODO)

Variablen-Deklaration

Mit einer Variablen-Deklaration wird eine Variable im aktuellen Gültigkeitsbereich erzeugt, d.h. als globale Variable oder lokale Funktionsvariable. Sie kann entweder ohne Wert instantiiert werden:

```
var alter;
```

Oder der Anfangswert kann gleich angegeben werden:

```
var alter = 32;
```

Anstelle der `32` kann ein beliebiger Ausdruck stehen. Das Schema lautet also:

```
var Bezeichner = Ausdruck;
```

Kontrollstrukturen: Verzweigungen und Schleifen

Kontrollstrukturen sind Anweisungen, die wiederum Blöcke mit Anweisungen enthalten. Dadurch ist es möglich, die Ausführung der in den Blöcken enthaltenen Anweisungen entsprechend bestimmter Regeln zu kontrollieren.

Man unterscheidet grundsätzlich zwei Arten von Kontrollstrukturen: Verzweigungen und Schleifen. Mittels Verzweigungen ist es möglich, die Ausführung einer oder mehrer Anweisungs-Blöcke von Bedingungen abhängig zu machen. Schleifen ermöglichen, einen Anweisungs-Block wiederholt ausführen zu lassen.

Verzweigungen

Zu den Verzweigungen gehört die bedingte Anweisung, auch `if`-Anweisung genannt.

```
if (alter >= 18) {
   alert("Volljährig!");
} else {
   alert("Noch nicht volljährig.");
}
```

Zwischen den runden Klammern steht ein beliebiger Ausdruck, zwischen den geschweiften Klammern eine oder mehrere Anweisungen. Das allgemeine Schema lautet:

```
if (Ausdruck) {
   Anweisungen
} else {
```

```
   Anweisungen
}
```

Mit der `switch`-Anweisung lassen sich Verzweigungen notieren, bei denen ein Wert mit vielen anderen Werten verglichen wird und eine entsprechende Anweisung ausgeführt wird.

```
switch (alter) {
   case 10 :
      alert("zehn");
   case 20 :
      alert("zwanzig");
}
```

Schleifen

Ist zur Laufzeit die Anzahl der Wiederholungen bereits beim Eintritt in die Schleife bekannt, verwendet man häufig die `for`-Anweisung.

```
for (var i = 0; i < 5; i++) {
    alert("Sie müssen noch " + (5 - i) + " mal klicken.");
}
```

Beim Schleifeneintritt wird zunächst die Variable `i` deklariert und mit dem Wert `0` initialisiert. Anschließend wird der Anweisungs-Block so oft ausgeführt, bis der Wert von `i` die Zahl 4 übersteigt. Da`i` nach jedem Schleifendurchlauf um eins erhöht wird, wird die Schleife nach fünf Durchläufen verlassen.

Allgemein wird die Ausführung einer `for`-Anweisung von drei Teilen bestimmt. Einmal der**Initialisierung**, in obigem Beispiel `var i = 0`. Hier wird im normalen Gebrauch eine so genannte**Schleifenvariable** mit einem Startwert initialisiert. Im zweiten Teil, der **Bedingung**, wird eine Eigenschaft der Schleifenvariable geprüft. Als drittes wird in der **Fortsetzung** angegeben, was nach jedem Durchlauf mit der Schleifenvariable getan werden soll.

Ausdruck-Anweisung (Expression-Statement)

Die meisten Anweisungen sind sogenannte Ausdruck-Anweisungen. Sie bestehen lediglich aus einem Ausdruck, und ein Ausdruck kann sehr unterschiedlich aussehen.

```
window.print();
alter = 18;
objekt.nummer = 5;
objekt.nummer++;
meldung = alter >= 18 ? "volljährig" : "noch nicht volljährig";
1 + 1;
```

Eine Ausdruck-Anweisung sollte immer mit einem Semikolon abgeschlossen werden.

Weitere Statements

break, return, continue usw.

Ausdrücke (Expressions)

Ein Ausdruck besteht aus Bezeichnern und Literalen, die mit Operatoren verknüpft werden

```
alter = 18
element.appendChild(element2)
```

```
objekt.nummer++
alert("Vorname: " + vorname)
```

Operatoren

Ein, zwei oder drei Operanden. Die Operanden können selbst als Ausdrücke notiert sein.

Bezeichner (Identifier)

siehe Kernobjekte

Literale

siehe Kernobjekte

Funktionsdeklarationen

siehe Funktionen

JavaScript: Objekte und Variablen

Objekte, Eigenschaften und Methoden

Ein Objekt ist grob gesagt ein Bündel von Informationen im Speicher. In JavaScript haben wir auf alle zugänglichen Informationen Zugriff als ein Objekt.

Ein Objekt funktioniert als eine Zuordnungsliste, die unter bestimmten Namen weitere**Unterobjekte**, auch **Member** genannt, speichert. Diese Unterobjekte teilt man in **Eigenschaften** und**Methoden**. Methoden sind ausführbare Funktionen, die dem Objekt zugehören, Eigenschaften sind alle nicht ausführbaren Unterobjekte.

Durch diese Verschachtelung von Objekten entsteht eine beliebig lange Kette von Objekten, die aufeinander verweisen.

Der Browser stellt einem Script eine große Menge von **vordefinierten Objekten** zur Verfügung. Ein Script nutzt einerseits diese vordefinierten Objekte, indem es Eigenschaften ausliest und Methoden aufruft. Andererseits definiert das Script eigene Objekte. Diese vom JavaScript-Programm selbst eingeführten Objekte werden üblicherweise **Variablen** genannt.

Konstruktoren, Prototypen und Instanzen

Ein JavaScript-Objekt gehört einem gewissen Typ an. Diese Zugehörigkeit stellt sich in JavaScript so dar, dass ein Objekt von einer Funktion erzeugt wurde. Diese erzeugende Funktion wird **Konstruktor**genannt. Die meisten Konstruktorfunktionen sind JavaScript-intern und vordefiniert, Sie können allerdings auch selbst definierte Funktionen als Konstruktoren verwenden.

Die Objekte, die ein bestimmter Konstruktor erzeugt hat, werden **Instanzen** (Exemplare, Abkömmlinge) dieses Konstruktors genannt. Beispielsweise ist ein String-Objekt (eine Zeichenkette) eine Instanz der Konstruktorfunktion `String`.

Jedes Objekt hat eine Eigenschaft namens `constructor`, die auf die Konstruktorfunktion verweist, die es hergestellt hat. Beispielsweise liefert `"Dies ist ein String".constructor` die Konstruktorfunktion `String`.

Ein Konstruktor ist eine Art Fabrik, die Instanzen nach immer gleichem Muster produziert. Dieses Muster, der Bauplan für die Instanzen ist selbst ein Objekt: Das sogenannte prototypische Objekt, kurz**Prototyp**. Instanzen sind so gesehen Kopien des Prototypen und werden ihm nachgebildet.

Das prototypische Objekt hängt an der Konstruktorfunktion. Jede Funktion hat eine Eigenschaft namens `prototype`, die auf das zugehörige prototypische Objekt verweist. `String.prototype`liefert beispielsweise das Objekt, das den Bauplan für alle String-Instanzen definiert.

Jedes Objekt stammt also von einem Konstruktoren ab, dem ein Prototyp zugeordnet ist. Wenn wir ein Objekt erzeugen, dann wird hinter den Kulissen die entsprechende Konstruktorfunktion aufgerufen. Die Instanz erbt alle Eigenschaften und Methoden des Prototypen, indem sie an das frisch erzeugte, bisher leere Instanz-Objekt kopiert werden. Dieser Vorgang wird **prototypische Vererbung** genannt.

Prototypen können auch untereinander erben. Auf diese Weise sind die Objekttypen voneinander ableitbar. Ein String-Objekt ist gleichzeitig eine Instanz vom `String`-Konstruktor wie auch eine Instanz vom `Object`-Konstruktor.

Das prototypische Objekt ist ein ganz normales JavaScript-Objekt, dem Sie neue Eigenschaften und Methoden hinzufügen können. Auf diese Weise können alle Instanzen, die vom fraglichen Prototypen erben, auf einen Schlag mit neuen Fähigkeiten ausgestattet werden. Diese Methode nennt sich **prototypische Erweiterung**.

Aus Sicht der JavaScript-Programmierung passieren diese Vorgänge meist unbemerkt. Dass beim Erstellen eines Objektes ein Konstruktor aufgerufen wird und der Prototyp als Vorbild für das Objekt dient, wird nur ersichtlich, wenn wir in diese Vorgänge eingreifen wollen oder eigene Konstruktoren und Prototypen schreiben. Dennoch ist das Wissen um Konstruktoren und Instanzen grundlegend, um den Aufbau der Objektwelt und die Objekttypen in JavaScript zu verstehen.

Klassen – gibt es nicht in JavaScript

Objekttypen und Vererbung werden in vielen anderen Programmiersprachen über Klassen gelöst. JavaScript hingegen kennt keine Klassen, sondern nur die besagten Konstruktorfunktionen und Prototypen. ...

Bezeichner: Objekte ansprechen

Bezeichner (engl. Identifier) sind Namen für Objekte. Um ein vorhandenes Objekt in einem JavaScript anzusprechen oder einen Wert unter einem Namen abzuspeichern, notieren wir dessen Namen.

```
alert(vorname);
```

Hier werden gleich zwei Bezeichner notiert: Einmal `alert` und einmal `vorname`. `alert`bezeichnet die vordefinierte globale Funktion, mit der sich Meldungen ausgeben lassen. `vorname` steht beispielhaft für eine durch das Script selbst erzeugte Variable.

Unterobjekte ansprechen

Um ausgehend von einem Objekt ein Unterobjekt (Member) anzusprechen, notieren wir den einen Objektnamen, dann einen `.` und danach den Namen des Unterobjekts. Schematisch:

```
objekt.unterobjekt
```

Der Punkt (`.`) ist der sogenannte *Property Accessor Operator* (engl. Operator zum Zugriff auf Eigenschaften).

Auf diese Weise können wir ganze Ketten an Objektnamen notieren, um das gewünschte Objekt zu fassen zu bekommen:

```
window.location.href.substring(0, 4)
```

Hier wird die `substring`-Methode des durch `window.location.href` referenzierten String-Objekts angesprochen, um das Protokoll `http` aus der Adresse (URL) des gegenwärtigen HTML-Dokuments zu extrahieren.

Eine alternative Methode zum Ansprechen von Unterobjekten ist die Klammer-Schreibweise. Dabei wird der Name des Unterobjektes zwischen eckigen Klammern als String notiert:

```
objekt["unterobjekt"]
```

Dies hat denselben Effekt wie `objekt.unterobjekt`. Der Vorteil dieser Schreibweise kommt zum Tragen, wenn der Objektname variabel ist und erst zur Laufzeit des Scriptes feststeht. Zwischen den Klammern lässt sich ein beliebiger Ausdruck notieren, also beispielsweise ein Bezeichner, der auf eine String-Variable verweist:

```
objekt[stringVariable]
```

Der JavaScript-Interpreter nimmt den String als Objektnamen und sucht nach einem entsprechenden Unterobjekt.

...

Konventionen für Bezeichner

JavaScript-Bezeichner unterliegen gewissen Regeln, die Sie beim Wählen von Objektnamen beachten müssen:

Ein Bezeichner darf nur aus Buchstaben, arabischen Ziffern (0-9), dem Dollarzeichen (`$`) sowie dem Unterstrich (`_`) bestehen. Jedes dieser Zeichen darf an beliebiger Stelle vorkommen, mit Ausnahme der Ziffern, welche nicht an erster Stelle stehen dürfen.

Alle anderen Zeichen (etwa Leer- oder Sonderzeichen) sind in Bezeichnern nicht erlaubt.

Beispiele für **erlaubte Bezeichner**:

```
mitarbeiter_name
mitarbeiterName
_mitarbeitername
$mitarbeitername
lohnIn$
mitarbeiter1
twenty4seven
```

Beispiele für **nicht erlaubte** Bezeichner:

```
mitarbeiter name
mitarbeiter-name
mitarbeiter.name
lohnIn€
lohnIn£
2raumwohnung
ein♥FürTiere
arbeiter&angestellte
```

Buchstaben sind all diejenigen Zeichen, die in der Unicode-Zeichendatenbank als »Letter«gekennzeichnet sind. Das sind eine ganze Menge: Neben dem lateinischen Alphabet gehören Buchstaben mit Diakritika wie ü, ö, ä, ß, é, â, ì, å, õ usw. dazu. Griechische, kyrillische, hebräische, japanische, chinesische Zeichen usw. sind ebenfalls erlaubt.

Allerdings beschränken sich die meisten JavaScript-Programmierer auf lateinische Buchstaben ohne diakritische Zeichen. Der Bezeichner `firmengröße` ist also möglich, üblicher ist allerdings,`firmengroesse` zu notieren, um etwa Schwierigkeiten bei der Zeichenkodierung aus dem Weg zu gehen.

Bracket Notation erlaubt beliebigen String

Funktionen und Methoden aufrufen

...

Call-Operator

funktion(parameter)

objekt.methode(parameter)

Das globale Objekt `window`

Variablen

Gültigkeitsbereich (Scope)

Identifier Resolution

Globale Variablen

Lokale Variablen (Funktionsvariablen)

Objekte erzeugen: Literale und Instantiierung

Objekte können in JavaScript auf zwei Weisen erzeugt werden: Durch einen **Literal** oder eine ausdrückliche **Instantiierung**.

Ein Literal (englisch für *wörtlich*, *buchstäblich*) ist eine Kurzschreibweise, mit der Sie Werte am schnellsten notieren können. `1.23` ist ein Literal, der ein Objekt vom Typ `Number` erzeugt. `"Hallo Welt!"` ist ein `String`-Literal, `true` und `false` sind `Boolean`-Literale.

Die Langschreibweise erzeugt ausdrücklich eine Instanz eines Kernobjekts, indem es den Konstruktoren aufruft. Wenn wir z.B. ein Objekt vom Typ `Array` erzeugen wollen, können wir `new Array()` schreiben.

Nicht alle Objekttypen lassen sich auf beide Weisen erzeugen und das Ergebnis ist auch nicht immer dasselbe (siehe Objekte und primitive Typen). Wenn eine Literalschreibweise für den gewünschten Objekttyp existiert, dann sollten Sie diese nutzen. Für manche Objekttypen existiert keine Literalschreibweise, sodass Sie beispielsweise `new Date()` notieren müssen.

Kernobjekte und Datentypen

1. Einführung
2. Die Mutter aller Objekte: Object

Einführung

Die Kernobjekte in JavaScript repräsentieren in erster Linie die grundlegenden **Datentypen**, z.B. Zahlen oder Zeichenketten (sogenannte *Strings*). Datentypen sind in JavaScript objektorientiert gelöst: Jedes Objekt ist eine Instanz einer bestimmten Konstruktorfunktion. Von dessen Prototypen erbt die Instanz Eigenschaften und Methoden.

...

Die Mutter aller Objekte: Object

Alle Objekte stammen von einem obersten Kernobjekt ab: `Object`. Jedes Objekt ist demnach zumindest vom allgemeinen Typ `Object`, darüber hinaus kann es einem spezifischeren Typ angehören (z.B. `String`).

Ein Objekt vom Typ `Object` ist bloß ein Container für weitere Daten [... doppelt]

Object-Objekte als Zuordnungslisten (Hashes)

Als grundlegender Typ hinter allen anderen Typen interessiert uns `Object` nur wenig. Diejenigen Objekte, die nur von `Object` abstammen, sind jedoch nützlich, wenn man in JavaScript eine Zuordnungsliste benötigt oder andere, zusammenhängende Objekte geordnet abspeichern will. Eine einfache Anwendung von `Object` könnte so aussehen:

```
var adresse = new Object();
adresse.name = "Max Mustermann";
adresse.straße = "Königsallee 56";
adresse.stadt = "Frankfurt";
```

Solche Objekte sind für die Strukturierung von Daten oder sogar JavaScript-Programmen selbst äußerst nützlich. Denn in solchen Objekten lassen sich nicht nur Zeichenketten oder Zahlen unter bestimmten Namen abspeichern, sondern auch beispielsweise Funktionsobjekte. Und sie lassen sich beliebig verschachteln.

Object-Literale

Das obige Beispiel verwendet die Langschreibweise `new Object()` zum Erzeugen eines`Object`-Objektes. Anschließend werden dem leeren Objekt Eigenschaften hinzugefügt. Es gibt allerdings auch eine kurze Literalschreibweise, die ungemein einfacher und verbreiteter ist. Um ein solches **Object-Literal** zu notieren, gibt es einen bestimmten Ausdruck: Er fängt mit einer öffnenden geschweiften Klammer an und endet mit einer schließenden. Dazwischen werden die Eigenschaften mit Name und Wert aufgelistet. Zwischen Name und Wert steht ein Doppelpunkt, zwischen den Eigenschaften ein Komma. Das Schema sieht demnach so aus:

```
{
   name1 : wert3,
   name2 : wert2,
   name3 : wert3
```

```
}
```

Die Anzahl der notierten Eigenschaften ist nicht begrenzt. Zu beachten ist, dass hinter der letzten Zuweisung kein Komma notiert wird.

Das erste Beispiel sieht in der Literalschreibweise so aus:

```
var adresse = {
   name : "Max Mustermann",
   straße : "Königsallee 56",
   stadt : "Frankfurt"
};
```

Dies erzeugt haargenau das gleiche Objekt wie die Langschreibweise.

String, Number und Boolean

String, Number und Boolean sind die wichtigsten einfachen Datentypen. Sie haben einen direkt und eindeutig darstellbaren Wert.

- Ein String-Wert ist eine Zeichenkette. Darin können Sie einzelne Zeichen bis hin zu ganzen Texten speichern.
- Ein Number-Wert ist eine Zahl. Es kann eine Ganzzahl oder eine Kommazahl sein, positiv oder negativ.
- Ein Boolean-Wert drückt einen Wahrheitswert aus. Dieser kann zwei Zustände annehmen: `true`(wahr) oder `false` (falsch).

```
var zahl = 1.23;
var zeichenkette = "Hallo Welt!";
var boolean = true;
```

Diese Anweisungen definieren eine `String`-, eine `Number`- und eine `Boolean`-Variable mithilfe der jeweiligen Literalschreibweise.

...

Objekte und primitive Typen

Zwar verhalten sich alle Werte in JavaScript in gewissen Situationen wie Objekte, aber strenggenommen gibt es eine Unterscheidung zwischen vollwertigen Objekten und sogenannten einfachen Werten, im der englischen Fachsprache **Primitive Values**, kurz **Primitives** genannt.

Diese Doppelung betrifft die eben behandelten `Boolean`-, `Number`- und `String`-Werte. Diese können nämlich entweder als Primitive oder als vollwertiges Objekt notiert werden:

- Die Literalschreibweise `var string = "Hallo Welt"` erzeugt ein *String-Primitive*.
- Die Instantiierung `var string = new String("Hallo Welt");` erzeugt ein *String-Objekt*.

Dieser Unterschied macht sich an zwei Stellen bemerkbar:

Referenzen und Kopien

Primitives werden **als Kopie** an Funktionen übergeben, während Objekte **als Referenzen** auf dieselbe Speicherstelle (in anderen Programmiersprachen »Zeiger« genannt) übergeben werden.

Gegeben ist folgender Fall: Sie notieren ein Objekt als Variable. Dieses Objekt übergeben Sie einer Funktion und in der Funktion nehmen Sie Änderungen am Objekt vor, fügen ihm z.B. eine Eigenschaft hinzu.

Wenn das Objekt als *Referenz* übergeben wird, dann haben Sie nach dem Funktionsaufruf auf das geänderte Objekt Zugriff. Denn

an beiden Stellen, innerhalb und außerhalb der Funktion, haben Sie Zugriff auf ein und dasselbe Objekt.

Wenn das Objekt als *Primitive* übergeben wird, dann haben Änderungen daran keine Auswirkung auf die Variable im ursprünglichen Kontext – es sei denn, die Funktion gibt einen Primitive zurück und Sie arbeiten mit dem Rückgabewert der Funktion weiter.

Gleichheit und Identität

Der Vergleichsoperator `==` ergibt beim Vergleich zweier vollwertiger Objekte nur dann `true`, wenn es sich um ein und dasselbe Objekt handelt. Er verhält sich in dem Fall wie der Identitätsoperator`===`. Zwei Objekte können also niemals gleich sein, es sei denn, sie sind identisch.

Der Vergleich `new String("Hallo Welt") == new String("Hallo Welt")` ergibt`false`, denn die beiden String-Objekte sind nicht identisch. Bei Primitives hingegen gibt es eine Gleichheit unabhängig von der Identität: `"Hallo Welt" == "Hallo Welt"` ergibt erwartungsgemäß`true`.

Empfehlung

Sofern Sie keinen besonderen Grund haben, sollten Sie `Boolean`-, `Number`- und `String`-Werte stets als Primitives, also mit der Literalschreibweise notieren.

...

Function (Funktionsobjekte)

Funktionen sind JavaScript-Objekte, die vom Konstruktor `Function` abstammen. Ein JavaScript-Programm ist üblicherweise in verschiedene Funktionen unterteilt, die einander aufrufen. Eine Funktion gruppiert zusammengehörige Anweisungen und löst eine gewisse isolierbare Teilaufgabe. Anstatt denselben oder sehr ähnlichen Code immer wieder zu notieren, notiert man stattdessen eine Funktion, die mehrfach aufgerufen werden kann. Durch sogenannte Parameter können ihr variable Informationen bei jedem Aufruf mitgeteilt werden.

Näheres zu Funktionen finden Sie im eigenen Abschnitt Funktionen.

Array: Geordnete Listen

Array sind numerische geordnete Listen mit anderen Objekten. Immer wenn mehrere gleichförmige Objekte in einer bestimmten Abfolge gespeichert werden sollen, sind Arrays die passende Struktur.

```
var städte = [ "Berlin", "Köln", "Hamburg", "München", "Frankfurt" ];
var lottozahlen = [ 4, 12, 23, 33, 42, 44 ];
var mitarbeiter = [
   {
      name : "Margarethe",
      geburtsdatum : new Date(1972, 4, 12),
      durchwahl : 401
   },
   {
      name : "Michael",
      geburtsdatum : new Date(1962, 11, 2),
      durchwahl : 402
   },
   {
      name : "Monika",
      geburtsdatum : new Date(1958, 5, 25),
      durchwahl : 403
   }
];
```

RegExp: Reguläre Ausdrücke

Mit regulären Ausdrücken lassen sich Muster für Zeichenabfolgen notieren, mit deren Hilfe sich Texte durchsuchen und automatisierte Ersetzungen vormehmen lassen. Mit einem regulären Ausdruck kann beispielsweise geprüft werden, ob ein String in einer bestimmten Weise aufgebaut ist und somit einer Konvention entspricht.

Wenn ein Formularfeld nur eine ganze Zahl enthalten darf, dann lässt sich mit einem regulären Ausdruck testen, ob das Feld tatsächlich nur Ziffern enthält. Enthält es noch andere Zeichen oder einen Komma-Wert, so kann der Wert vor dem Absenden des Formulars automatisch korrigiert werden.

Reguläre Ausdrücke sind eigene Objekte vom Typ `RegExp`. Um ein solches Objekt zu erzeugen, können wir einen RegExp-Literal notieren oder `new RegExp()` aufrufen. ...

Date: Datumsobjekte

...

Math: Mathematische Hilfsmethoden

...

Objekttypen außerhalb des Kerns

DOM-Knoten, Elementobjekte, ...

JavaScript: Funktionen

Einführung

Mit Funktionen können Sie flexible Teilprogramme notieren.

... Das haben Sie sicher schon im Grundlagenkurs Programmierung gelernt, oder kennen es schon von anderen Programmiersprachen. ...

Die vorgegebenen JavaScript-Objekte bieten in erster Linie Funktionen, die Sie in Ihren Scripten aufrufen können.

In JavaScript spielen Funktionen einen höheren Stellenwert als in anderen Programmiersprachen und haben einige

Besonderheiten - deshalb spricht man davon, dass JavaScript Aspekte einer*funktionalen Programmiersprache* besitzt. In JavaScript sind Funktionen nicht einfach feste, einmal definierte Script-Bestandteile, sondern selbst Objekte, mit denen man im Script nahezu ungehindert arbeiten kann. Sie können sogar problemlos neue Funktionen zur Laufzeit erzeugen und einmal definierte wieder löschen.

Wenn Sie **eigene Funktionen definieren**, so können Sie sie an bestimmte andere Objekte hängen. Das bedeutet, sie als Unterobjekte eines Objektes zu speichern. In diesem Fall spricht man in der Terminologie der objektorientierten Programmierung von einer **Methode**. In anderen Fällen ist es sinnvoller, die Funktion als lokale Variable in einer anderen Funktion anzulegen. Und schließlich brauchen Sie die Funktion gar nicht zu speichern - sie können Sie auch anonym (namenlos) anlegen, nur um sie z.B. als Parameter an eine andere Funktion weiterzugeben. Diese drei Möglichkeiten werden Ihnen später noch klarer werden, wenn wir betrachten, auf welche Weise Sie Funktionen erzeugen können.

Funktionsdeklarationen

Es gibt drei Arten, wie Sie Funktionen notieren und damit Funktionsobjekte erzeugen können. Die einfachste und wichtigste Art ist die sogenannte Funktions-Deklaration (auf englisch *function declaration*). Deren Syntax ist folgendermaßen aufgebaut:

```
function Funktionsname (Parameterliste) {
Anweisungen
}
```

Der **Funktionsname** muss den üblichen Anforderungen an JavaScript-Bezeichner genügen: Sie dürfen Buchstaben, Zahlen und einige Sonderzeichen (das Dollar-Zeichen, den Unterstrich ... TODO) verwenden. Der Funktionsname darf in diesem Fall aber keine Leerzeichen enthalten. TODO: Wann darf er das?

Die **Parameterliste** zwischen den beiden runden Klammern ist eine durch Kommas getrennte Liste von Namen. Für diese gelten die besagten Namenskonventionen. Unter den in dieser Auflistung vergebenen Namen können Sie innerhalb der Funktion auf die übergebenen Parameter zugreifen.

Zwischen den beiden geschweiften Klammern wird der sogenannte Funktionskörper notiert: Darin werden die Anweisungen untergebracht, die beim Aufruf der Funktion ausgeführt werden sollen.

Folgendes Beispiel soll das Schema verdeutlichen:

```
function statusMeldung (meldungsTyp, meldungsText) {
var ausgabeElement = document.getElementById("meldungsausgabe");
ausgabeElement.className = meldungsTyp;
ausgabeElement.innerHTML = meldungsText;
}
statusMeldung("fehler", "Beim Absenden Ihrer Nachricht ist ein Fehler aufgetreten. " +
"Bitte versuchen Sie es erneut.");
```

Das Beispiel definiert eine Funktion mit dem Namen `statusMeldung`. Die Funktion erwartet zwei Parameter mit dem Namen `meldungsTyp` bzw. `meldungsText`.

Die Funktion wird nach dem Schema *`Funktionsname(Parameterliste)`* aufgerufen. Die beiden runden Klammern nach dem Funktionsnamen sind der eigentliche

...

Was passiert: wenn Sie direkt in einem script-Element notieren, wird eine globale Funktion angelegt. Das ist eine Methode des window-Objektes ... Globale Variablen sind wiederum nichts anderes als Eigenschaften des Objektes `window`.

bla() = window.bla()! window["bla"]!

Funktionsparameter

Bei der Deklaration weisen Sie den Funktionsparametern

```
function Summe (zahl1, zahl2, zahl3) {

}

Summe(5, 10, 15);
```

Variable Parameterzahl: Der arguments-Array

Rückgabewert (Ergebnis)

return

Lokale Variablen (Funktionsvariablen)

Gültigkeitsbereich (Scope)

Funktionsausdrücke

Wie gesagt gibt es neben der oben vorgestellten zwei weitere, also insgesamt drei Schreibweisen, mit denen Sie Funktionen erzeugen können. Je nach Verwendungszweck können die folgenden weniger bekannten Schreibweisen passender sein.

Die zweite Art, wie Sie Funktionen notieren können, ist der sogenannte **Funktions-Ausdruck**(englisch *function expression*). Diese Schreibweise hat viele Vorteile, Sie werden sie schätzen lernen und vielfältig anwenden können.

Um den Unterschied zwischen Funktionsdeklaration und Funktionsausdruck zu verstehen, müssen Sie den Unterschied zwischen *Anweisungen* (Statements) und *Ausdrücken* (Expressions) kennen. Die vorgestellte Funktionsdeklaration ist nämlich eine Anweisung, der Funktionsausdruck hingegen ein Ausdruck. Damit haben beide unterschiedliche Anwendungsmöglichkeiten. Sie können Funktionsausdrücke an viel mehr Stellen notieren als eine Funktionsdeklaration.

Das Schema eines Funktionsausdruckes sieht folgendermaßen aus:

```
function (Parameterliste) { Anweisungen }
```

Ein solcher Funktionsausdruck selbst ergibt lediglich eine Funktion, speichert Sie aber nicht unter einem Namen. Man spricht daher auch von **anonymen (namenlosen)** Funktionen.

Das Ergebnis des Ausdruckes, ein Funktionsobjekt, können Sie jedoch weiterverwenden. Beispielsweise können Sie das erzeugte Funktionsobjekt in einer Variable speichern:

```
var Funktionsname = function (Parameterliste) { Anweisungen };
```

Dieses Variablenzuweisung mit Funktionsausdruck hat denselben Effekt wie die klassische Funktionsdeklaration `function Funktionsname (...) {...}`. Das bedeutet, sie sind unter allen Umständen austauschbar.

Darüber lässt sich auch genauer verstehen, was eine Funktionsdeklaration macht. Wenn die gleichwertigen Anweisungen innerhalb einer Funktion notiert werden, wird eine *lokale Variable* erzeugt, in der die neue Funktion gespeichert wird. Stehen sie außerhalb einer Funktion ausgeführt, dann wird eine *globale Variable* erzeugt, das heißt die neue Funktion wird als Methode von Objektes `window`angelegt.

Was sind nun die **Vorteile** eines Funktionsausdruckes?

Mit Funktionsdeklarationen erzeugt man üblicherweise globale Funktionen (`window`-Methoden). Wenn Sie eine Funktion mittels Funktionsausdruck erzeugen, müssen Sie diese nicht zwangsläufig global als `window`-Methode abspeichern, sondern können sie auch an einem anderen Objekt speichern. Auf diese Weise können Sie Ordnung in Ihre Scripte bringen und zusammenhörige Variablen z.B. unter einem Objekt gruppieren. Ein Beispiel:

```
var bildergalerie = new Object();
bildergalerie.abspielen = function () {
/* ... */
};
bildergalerie.abspielen();
```

Im obigen Beispiel wird eine leere Object-Instanz erzeugt, die als globale Variable mit dem Namen `bildergalerie` gespeichert wird. In der zweiten Zeile wird dem zunächst leeren Objekt eine Methode hinzugefügt. Die entsprechende Funktion wird mithilfe eines Funktionsausdrucks notiert. Das entstehende Funktionsobjekt wird in der Eigenschaft `abspielen` gespeichert (siehe Object-Objekte als Zuordnungslisten). In der dritten Zeile schließlich wird diese Funktion aufgerufen.

Die Gruppierung unter dem Objekt `bildergalerie` hat den Vorteil, dass der globale Gültigkeitsbereich, das `window`-Objekt, nicht übermäßig mit eigenen Objekten beschrieben wird. Der Verzicht auf globale Variablen hat den Vorteil, dass mehrere Scripte problemlos zusammenarbeiten können. [TODO: Diese Programmiertechnik der Kapselung zentral beschreiben.]

Im Beispiel wird lediglich das Objekt `bildergalerie` global gespeichert, das heißt als Eigenschaft von `window`. Folglich darf an keiner anderen Stelle eine gleichnamige globale Variable erzeugt werden, sonst würde das Objekt überschrieben werden. Die Funktion `abspielen` hängt hingegen als Methode am `bildergalerie`-Objekt. Sie kann anderen, gleichnamigen Funktionen nicht in die Quere kommen.

...

Eine weitere häufige Anwendung von Funktionsausdrücken findet sich im **Event-Handling**. Um eine Handler-Funktionen zu notieren, können Sie herkömmliche Funktionsdeklarationen nutzen:

```
// Handler-Funktion mit Funktionsdeklaration notieren
function init () {
window.alert("Dokument ist fertig geladen!");
}
// Event-Handler registrieren
window.onload = init;
```

Im Beispiel wird eine globale Funktion namens `init` angelegt und daraufhin als Event-Handler für das `load`-Ereignis beim `window`-Objekt registriert (siehe traditionelles Event-Handling).

Diese Schreibweise ergibt Sinn, wenn Sie die Funktion `init` später noch einmal benötigen. Üblicherweise ist das nicht der Fall: Man braucht solche Funktionen nur an der Stelle, wo man sie als Handler registriert; es ist nicht nötig, sie irgendwo unter einem Namen zu speichern.

In diesem Fall kann ein Funktionsausdruck den Code vereinfachen. Notieren Sie die Handler-Funktion mit einem Ausdruck und speichern Sie sie direkt in der `onload`-Eigenschaft:

```
window.onload = function () {
window.alert("Dokument ist fertig geladen!");
};
```

Dasselbe Prinzip können Sie überall beim Event-Handling anwenden. Wenn Sie beispielsweise einem Element einen `click`-Handler zuweisen wollen, so könnten Sie die fragliche Funktion mit einer Deklaration notieren:

```
function klickHandler () {
window.alert("Element wurde geklickt!");
}
document.getElementById("bla").onclick = klickHandler;
```

Üblicherweise besteht keine Notwendigkeit, die Handler-Funktion global unter dem Namen »klickHandler« zu speichern.

Stattdessen können Sie einen Funktionsausdruck verwenden und die erzeugte Funktion direkt als `click`-Handler abspeichern:

```
document.getElementById("bla").onclick = function () {
window.alert("Element wurde geklickt!");
};
```

Es gibt noch viele weitere Fälle, in denen das Zwischenspeichern einer Funktion, wie es eine Funktionsdeklaration zwangsläufig tut, unnötig ist - und damit gibt es es zahlreiche weitere Anwendungsmöglichkeiten für Funktionsausdrücke. Im Abschnitt über Closures werden wir darauf zurückkommen.

Function-Konstruktor

Wenden wir uns der dritten und letzten Möglichkeit zur Erzeugung von Funktionen zu. Dieser brauchen Sie keine große Aufmerksamkeit schenken, denn ihr Anwendungsbereich ist klein und ihr Gebrauch entsprechend selten.

Alle Funktionen, egal wie sie erzeugt wurden, sind Instanzen des `Function`-Konstruktors. Sie können daher auch direkt diesen Konstruktor aufrufen, um eine weitere Instanz zu erzeugen. Die Schreibweise lautet folgendermaßen:

```
new Function("Anweisungen", "Parametername1", "Parametername2", ...)
```

Sie rufen Function mit dem Schlüsselwort `new` auf. Der Konstruktor erwartet die Anweisungen, d.h. den Funktionskörper, im ersten Parameter. Dabei muss es sich um einen **String** handeln! Der zweite, dritte und alle folgenden Parameter enthalten die Parameternamen der neuen Funktion - ebenfalls als Strings. Wenn die zu erzeugende Funktion drei Parameter namens ...

Der Aufruf von `new Function(...)` erzeugt lediglich eine Funktion, speichert sie selbst aber noch nicht. Sie kennen das bereits vom Funktionsausdruck. Möchten Sie die erzeugte Funktion in einer lokalen Variable speichern, können Sie notieren:

```
var quadrat = new Function(
// Funktionskörper mit Anweisungen
"window.alert('Das Quadrat der Zahl ' + zahl + 'lautet: ' + (zahl * zahl);",
// Name des ersten Parameters
"zahl"
);
// Aufruf der erzeugten Funktion
quadrat(5);
```

Diese recht umständliche Notationsweise macht nur dann Sinn, wenn Sie in Ihrem JavaScript-Programm JavaScript-Code als String gespeichert haben und eine Funktion daraus machen wollen. Dies kommt freilich nur in einigen Spezialanwendungen vor.

Verwenden Sie nach Möglichkeit die beschriebenen Funktionsdeklarationen und -ausdrücke.

Verschachtelte Funktionen (Closures)

Fortgesch

siehe Artikel

```
function meineFunktion () {
document.getElementBy
}
window.setTimeout(meineFunktion, 5000);

function irgendeineFunktion () {
var bla = "string"; // diese Variable ist nur in dieser Funktion verfügbar
var status = 50;
```

```
setTimeout(
  function () {
        // Verschachtelte Funktion
        // Closure!
        alert(status);
  },
  5000
);
}
```

Funktionale Programmierg: Funktionen als Objekte verwenden

Funktionen sind ganz normale Objekte mit Eigenschaften und Methoden

Event-Handler und Callback-Funktionen

Funktionale Programmierung

Kontext einer Funktion: Das Schlüsselwort this

Kontext mit call und apply beeinflussen

JavaScript: Arbeitstechniken und Entwicklerwerkzeuge

Einleitung

Wenn Sie in die JavaScript-Entwicklung einsteigen, sollte Sie sich mit einigen **Werkzeugen**vertraut machen, die Ihnen das Schreiben eines Scriptes vereinfachen. Um zu einem funktionsfähigen Script kommen, um Fehler zu finden und um Script-Abläufe zu verstehen, gibt es einige bewährte**Arbeitsmethoden** und Fertigkeiten.

Erste Schritte mit Firefox, Firebug und einem Editor

JavaScripte können Sie mit ganz unterschiedlichen Programmen und Hilfsmitteln entwickeln und jeder Webautor hat unterschiedliche Bedürfnisse und Gewohnheiten. Sobald Sie einige Erfahrung gesammelt haben, sollten Sie sich umschauen, welche Arbeitsweise Ihnen am besten zusagt. Für den Anfang sei hier aber eine einseitige und parteiische Empfehlung ausgesprochen.

Firefox

Wenn Sie browserübergreifende JavaScripte für das öffentliche Web programmieren, so sollten Sie Ihre Scripte zunächst mit dem Browser **Mozilla Firefox** testen und sie mit dessen Hilfe fortentwickeln. Dieser Browser ist weit verbreitet, läuft auf allen relevanten Plattformen und ist einer der Ziel-Browser, auf dem Ihre Scripte auf jeden Fall funktionieren sollten. Firefox verfügt über ausgezeichnete JavaScript-Fähigkeiten und gibt präzise und hilfreiche JavaScript-Fehlermeldungen. Sie können Firefox untergetfirefox.com herunterladen.

Firebug

Eine entscheidende Hilfe stellt das Firefox-Zusatzprogramm **Firebug** dar. Firebug ist nicht nur ein mächtiges JavaScript-Werkzeug, sondern hilft auch enorm bei der HTML- und CSS-Entwicklung - insbesondere durch den sogenannten *DOM-Inspektor*, mit dem Sie den Elementbaum des HTML-Dokuments sowie die zugehörigen JavaScript- und CSS-Eigenschaften ansehen können. Nachdem Sie Firefox installiert haben, können Sie das Firebug-Addon über die Firefox-Addon-Datenbank installieren.

Firebug gibt Lernenden einen wertvollen Einblick in das DOM, veranschaulicht den Aufbau von JavaScript-Objekten und macht die Zusammenarbeit zwischen HTML, CSS und JavaScript verständlich. Seien Sie neugierig und spielen Sie mit Firebug herum – es wird Ihr Verständnis der Webtechniken und insbesondere des *DOM Scripting* enorm verbessern.

Texteditor

Eine weitere Voraussetzung für die JavaScript-Programmierung ist natürlich ein **Texteditor**, mit dem Sie die JavaScripte schreiben. Auch in puncto Editoren haben Sie zwischen tausenden eine Auswahl. Für den Anfang brauchen Sie keinen Editor mit besonderen Fähigkeiten. Er sollte jedoch Syntax-Highlighting für JavaScript-Code sowie die Anzeige von Zeilen- und Spaltennummern beherrschen. Denn diese Features erweisen sich direkt als hilfreich. Ein brauchbarer einfacher Editor für Windows istNotepad++.

Mit diesem Dreiergespann bestehend aus einem komfortablen Texteditor, einem hochentwickelten Browser und einem Analyse-Werkzeug für JavaScripte können sie gut gewappnet in die Programmierung einsteigen.

Fehlermeldungen und Fehlerkonsolen

Der erste Anlaufpunkt beim Testen von Scripten in einem Browser ist die **JavaScript-Konsole** des Browsers. Darin werden Fehler aufgelistet, die bei der Ausführung aufgetreten sind. Eine solche Fehlermeldung besteht üblicherweise aus einer Fehlerbeschreibung, der Script-Adresse sowie der Zeilennummer.

In der JavaScript-Konsole finden sich in erster Linie **Ausnahmefehler** (englisch *exception*s). Ein solcher führt zum sofortigen Abbruch der Ausführung des Scriptes. Ein Ausnahmefehler tritt beispielsweise auf, wenn Sie auf eine nicht existente Variable zugreifen, eine nicht existente Methode aufrufen oder eine Objekteigenschaft ansprechen, obwohl es sich beim jeweiligen Wert nicht um ein Objekt handelt.

Die Browser unterscheiden sich darin, wie sie diese Fehlermeldungen darstellen und wie informativ diese sind. Die verständlichsten und präzisesten Fehlermeldungen liefert üblicherweise der Firefox-Browser.

Firefox-Fehlerkonsole und Firebug

Im Firefox können Sie die Meldungen über das Menü *Extras* → *Fehlerkonsole* erreichen:

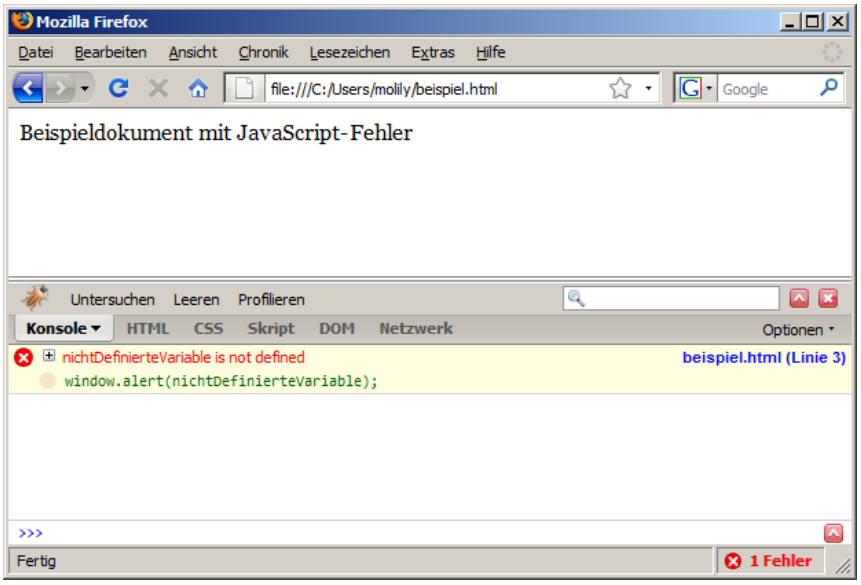
Der obige Screenshot zeigt eine beispielhafte Fehlermeldung, die auftritt, wenn auf eine nicht definierte Variable zugegriffen wird. Der Code, der diesen Fehler absichtlich ausgelöst hat, lautet:

```
window.alert(nichtDefinierteVariable);
```

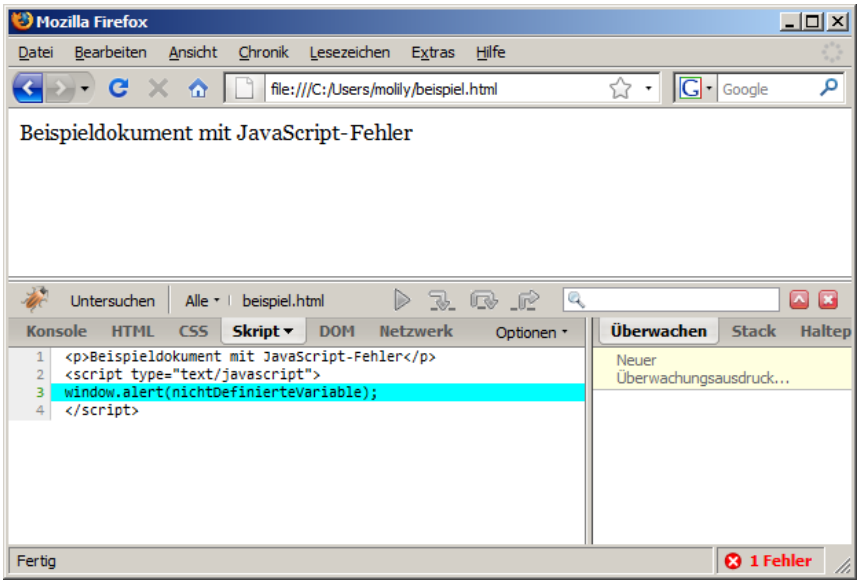
Wie der Fehlerkonsole zu entnehmen ist, steht dieser Code in der Zeile 3 des HTML-Dokuments mit der Adresse *file:///C:/Users/molily/beispiel.html*.

In der Fehlerkonsole von Firefox werden nicht nur JavaScript-Exceptions angezeigt, sondern auch andere Meldungen und Warnungen, die Firefox beim Verarbeiten von HTML- und CSS-Code ausgibt. Als Webentwickler sollten Sie daher immer ein Auge auf diese Fehlerkonsole haben.

Mithilfe des empfohlenen Firefox-Zusatz **Firebug** ist der Zugriff auf die Fehlerkonsole einfacher. Wenn bei der Ausführung eines Scriptes ein Ausnahmefehler auftritt, dann erscheint rechts unten in der Statusleiste ein rotes Fehlersymbol mit der Anzahl der Fehler. Wenn Sie darauf klicken, klappt Firebug am unteren Fensterrand auf und zeigt die Konsole samt Fehlermeldungen:



Firebug zeigt nicht nur die aus der herkömmlichen Firefox-Konsole bekannten Informationen an, sondern zeigt sogar die JavaScript-Codezeile, in der der Fehler aufgetreten ist. Außerdem führt ein Klick auf den grünen Code oder den blauen Zeilenverweis in den *Script*-Tab von Firebug. Darin wird der gesamte JavaScript-Code dargestellt und direkt zu der verantwortlichen Zeile gesprungen:



Nach Möglichkeit sollten Sie mit den JavaScript-Konsolen von Firefox bzw. dem Firebug-Zusatz arbeiten. Dennoch soll nicht verschwiegen werden, wie Sie in den anderen relevanten Browsern an die Fehlermeldungen kommen.

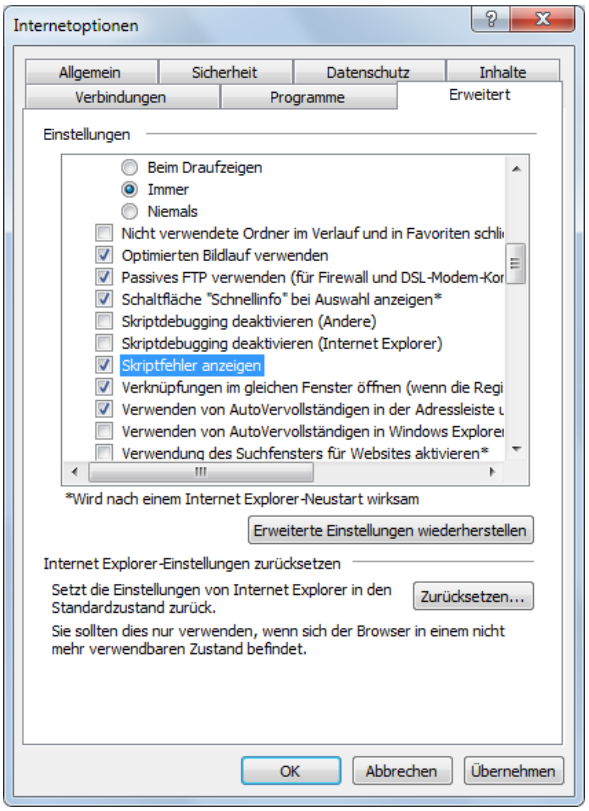
Fehlermeldungen im Internet Explorer ab Version 8

Der Internet Explorer hat zwei verschiedene MNodi, was den Umgang mit JavaScript-Fehlermeldungen angeht. Entweder zeigt er jeden JavaScript-Fehler in einem Dialogfeld an. Oder er versteckt diese Meldungen und macht sie nur auf Abruf zugänglich.

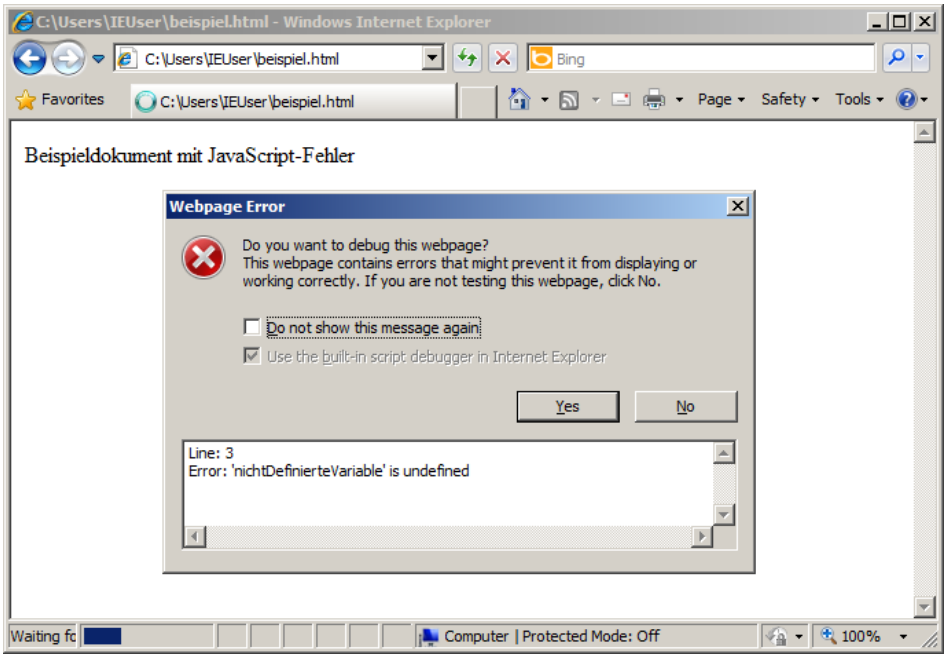
Wenn Sie Scripte im Internet Explorer testen, sollten Sie zumindest zeitweise die Anzeige jedes Fehlers aktivieren. Beim normalen Surfen auf fremden Sites hingegen stören diese Fehlermeldungen. Um die Meldungen zu aktivieren, öffnen Sie die *Internetoptionen* des IE. Diese finden Sie im Menü *Extras*. In dem sich öffnenden Fenster wählen Sie die Registerkarte *Erweitert*. In der dortigen Liste nehmen Sie folgende Einstellungen vor:

- *Skriptdebugging deaktivieren (Internet Explorer)*: Setzen Sie **kein** Häkchen in dieser Checkbox (d.h. Debugging soll aktiviert bleiben).
- Skriptfehler anzeigen: Setzen Sie ein Häkchen in dieser Checkbox (d.h. Skriptfehler sollen angezeigt werden).

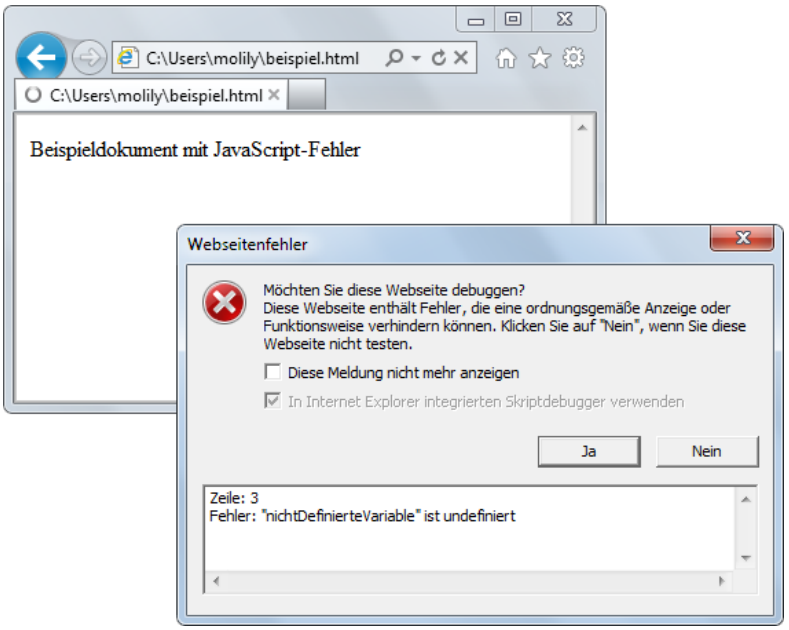
Der folgende Screenshot zeigt diese Einstellungen:



Verlassen Sie die Internetoptionen, indem Sie mit OK bestätigen. Wenn nun ein JavaScript-Fehler auftritt, öffnet sich automatisch ein solches Meldungsfenster. Dieses sieht im Internet Explorer 8 (englischsprachiges Windows) so aus:



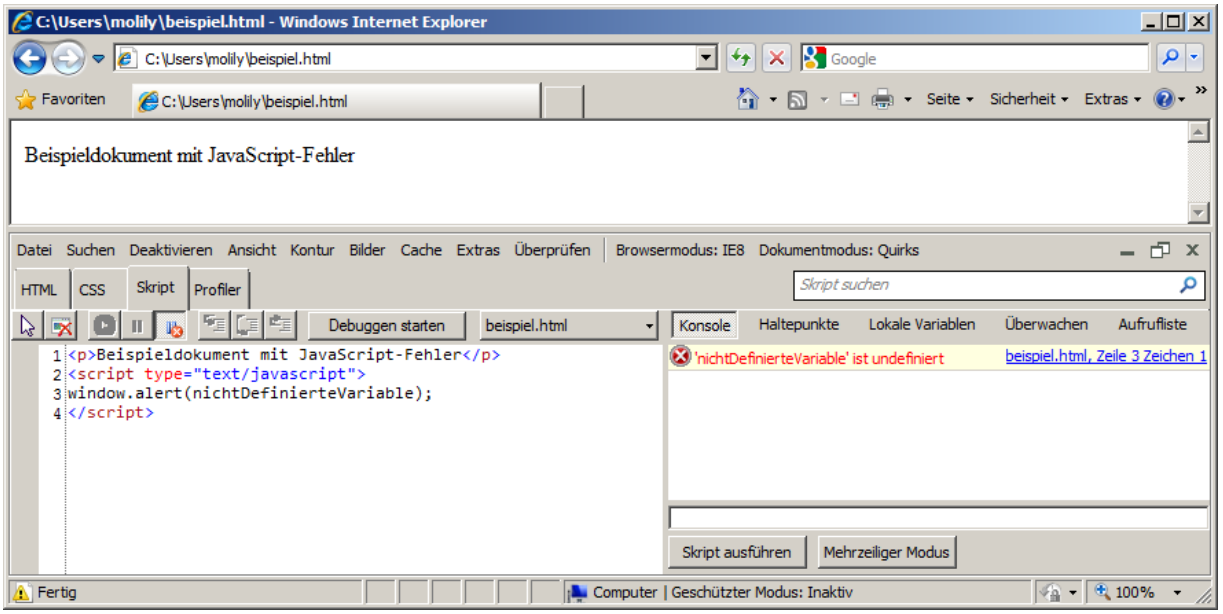
Der Internet Explorer 9 zeigt ein ähnliches Meldungsfenster an und bietet Ihnen ebenfalls an, den Fehler mit den Entwicklertools (siehe unten) zu debuggen:



Das Anzeigen von JavaScript-Fehlern ist nur während der Entwicklung sinnvoll. Sie sollten diese Möglichkeit nur solange in Anspruch nehmen, wie sie an JavaScripten arbeiten. Danach können sie die Fehlermeldungen in den Internetoptionen wieder ausschalten, damit Sie nicht beim normalen Surfen gestört werden.

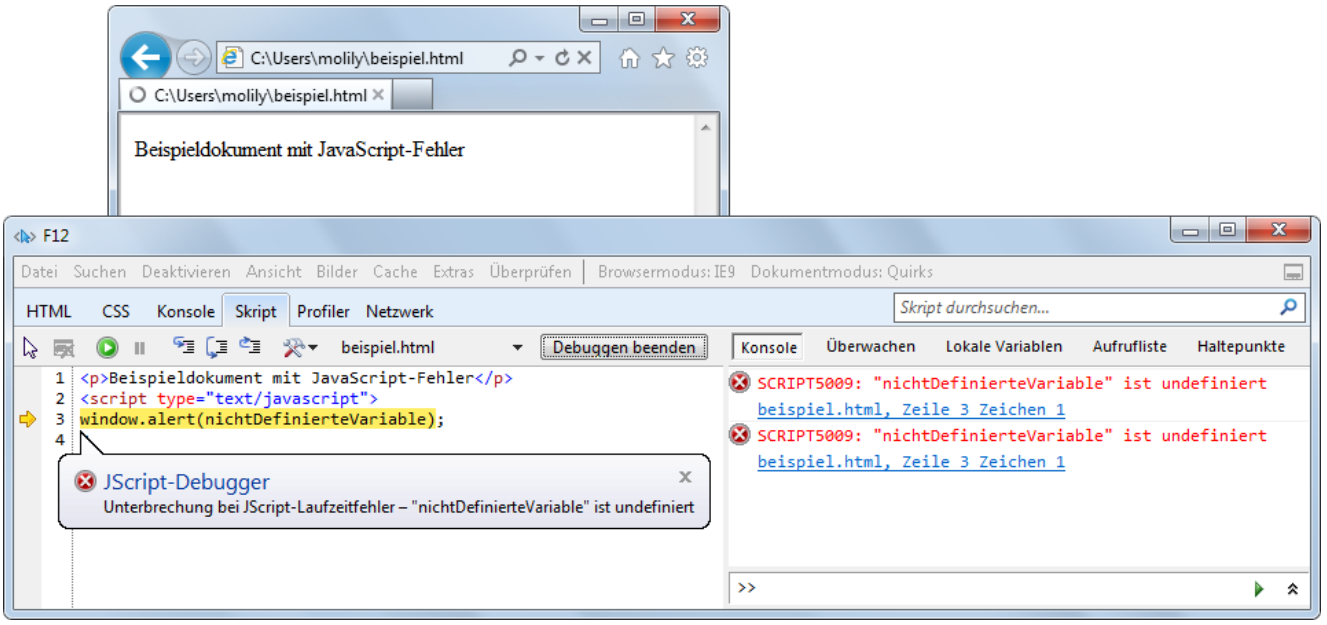
Internet Explorers: Fehlerkonsole der Entwicklertools

Neben diesen Meldungsfenstern kennt der Internet Explorer ab Version 8 eine JavaScript-Konsole, in der alle Fehlermeldungen mitgeloggt werden. Der IE besitzt dazu die sogenannten **Entwicklertools**, die stark an Firebug angelehnt sind. Beim Drücken der Taste `F12` klappt eine Leiste am unteren Fensterrand auf. Um JavaScript-Fehler zu sehen, müssen Sie den Tab *Skript* anwählen:



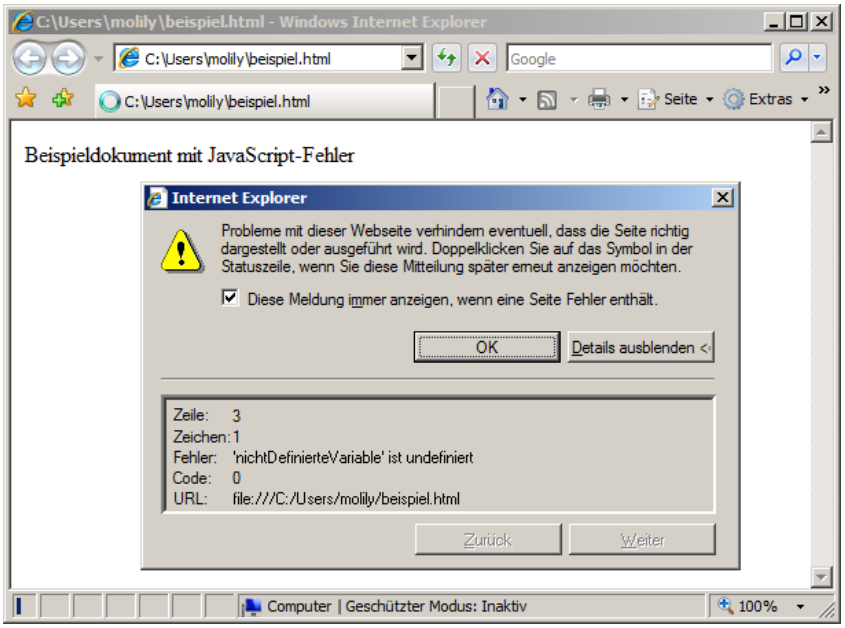
Diese Ansicht ist zweigeteilt: Im linken Bereich können alle eingebundenen JavaScripte angesehen werden, im rechten Bereich erscheint standardmäßig die JavaScript-Konsole. Wenn Ausnahmefehler auftreten, werden sie dort aufgelistet. Wie bei Firebug zeigen die IE-Entwicklertools neben dem Fehler ein Verweis auf die verantwortliche JavaScript-Codezeile an. Wenn Sie diesen anklicken, wird im linken Bereich die Codezeile hervorgehoben.

Sie können aus den Fehlermeldungsfenstern, welche im vorherigen Abschnitt gezeigt wurden, direkt in die Entwicklertools springen. Der Internet Explorer ab Version 8 bietet Ihnen an, den Debugger zu öffnen. Wählen Sie *Ja* bei der Frage *Wollen Sie die Seite debuggen?*, so öffnet sich automatisch der Skript-Tab der Entwicklertools. Darin wird die Codezeile markiert, in der der Fehler aufgetreten ist:



Fehlermeldungen in älteren Internet Explorern

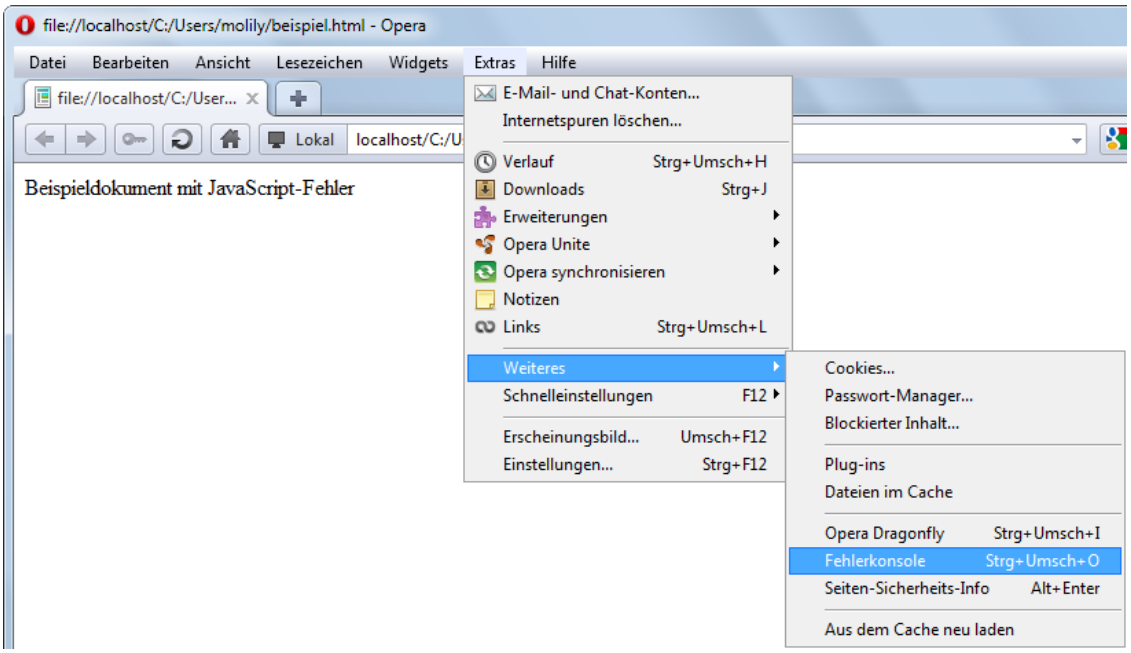
Sie sollten bei der JavaScript-Entwicklung die neueste Internet-Explorer-Version verwenden, wenn es nicht gerade um das absichtliche Testen mit älteren Versionen geht. Denn die Fehlermeldungen in älteren Versionen sind weniger präzise sowie oft unverständlich und verwirrend. In manchen Fällen wird die Zeilennummer nicht korrekt angegeben und bei ausgelagerten JavaScript-Dateien die tatsächliche Script-Adresse nicht angegeben, nur die Adresse des einbindenden HTML-Dokuments. Ältere Internet Explorer kennen keine Konsole, in der alle Fehlermeldungen mitgeloggt werden.



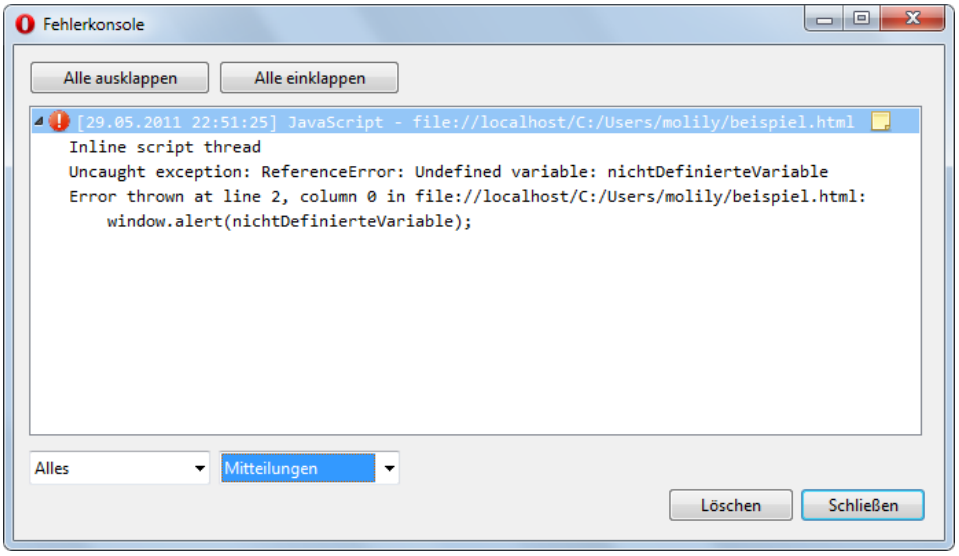
Ein komfortables Debugging von Scripten im Internet Explorer vor Version 8 ist daher ohne Zusatzprogramme schwer möglich. Ein mögliches Zusatzprogramm, das ein brauchbares Fehler-Logging anbietet, lautet Companion.JS bzw. DebugBar. Einen komplexeren JavaScript-Debugger bietet die umfangreiche Visual Web Developer 2010 Express Edition, die sich kostenlos herunterladen lässt.

Opera-Fehlerkonsole und Opera Dragonfly

Im Opera-Browser erreichen Sie die Fehlerkonsole über den Menüeintrag *Extras* → *Weiteres* → *Fehlerkonsole*. Vorher müssen Sie die volle Menüleiste gegebenenfalls erst einblenden über *Menü* → *Menüleiste zeigen*.



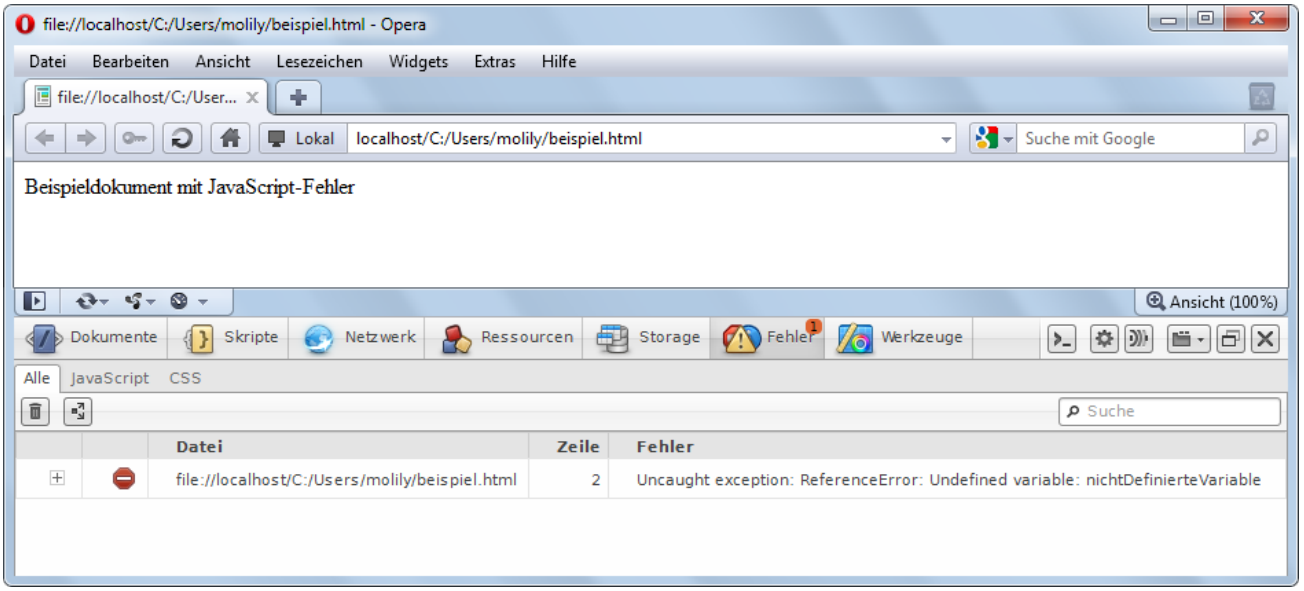
Die Fehlerbeschreibungen sind im Allgemeinen weniger verständlich als im Firefox, aber zeigen zumindest die Code-Zeile an, die den Fehler ausgelöst hat:



Ähnlich wie beim Firefox handelt es sich um eine Mehrzweck-Konsole, in der sich u.a. auch HTML- und CSS-Warnungen finden. Eine Filterung nach JavaScript-Fehlern ist jedoch möglich, indem Sie im linken Aufklappmenü *JavaScript* wählen.

Zudem verfügt Opera über einen Firebug-ähnlichen Zusatz namens *Dragonfly*, der über das Menü*Extras → Weiteres → Opera Dragonfly* aktiviert werden kann (das ist das oben abgebildete Menü).

Dragonfly besitzt eine integrierte Fehlerkonsole, die die Meldungen etwas übersichtlicher auflistet und ein Springen zum JavaScript-Code ermöglicht, der den Fehler ausgelöst hat. Zudem kann Dragonfly wie Firebug und die Entwicklertools des Interent Explorers am unteren Fensterrand angezeigt werden:

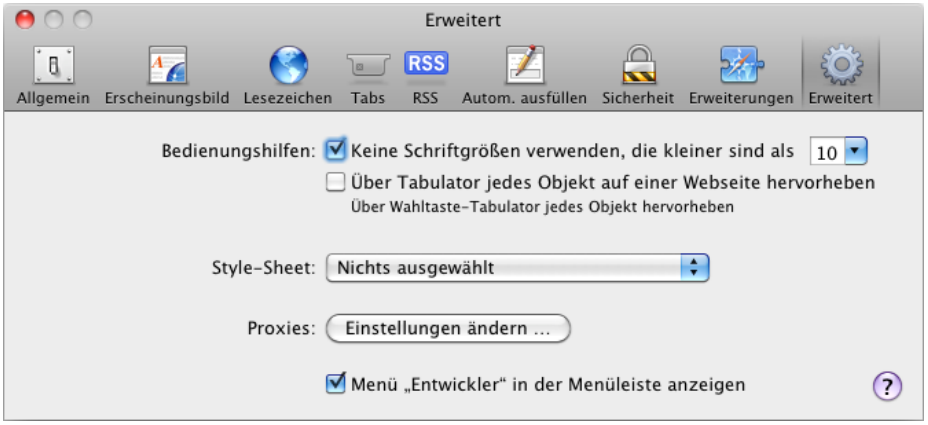


Fehlerkonsolen in Chrome und Safari

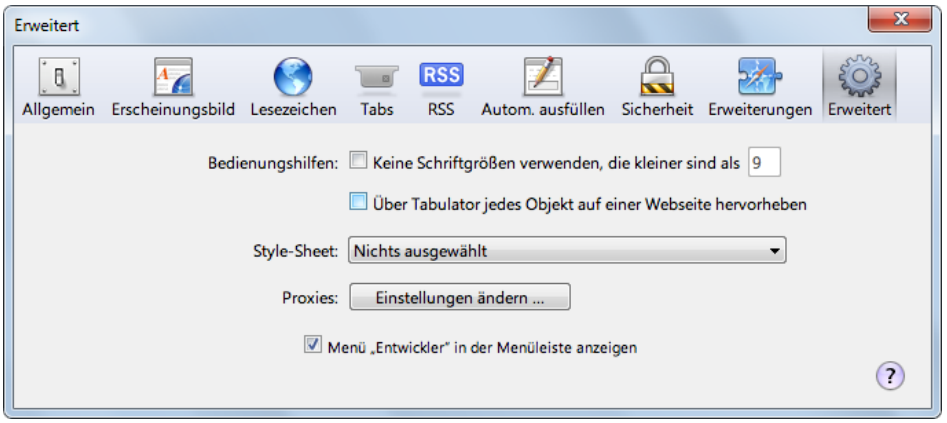
Google Chrome und Apple Safari verfügen über dasselbe Entwicklerwerkzeug, den *Web Inspector*, mit dem sich JavaScript sehr gut debuggen lässt..

Um im Safari an die Fehlermeldungen zu kommen, müssen Sie zunächst das sogenannte Developer-Menü aktivieren, denn darüber lässt sich die JavaScript-Fehlerkonsole öffnen. Dazu wählen Sie im Menü *Safari → Einstellungen…* (Mac OS) beziehungsweise *Bearbeiten → Einstellungen*(Windows). Unter der Registerkarte *Erweitert* aktivieren Sie die Checkbox *Menü „Entwickler“ in der Menüleiste einblenden*.

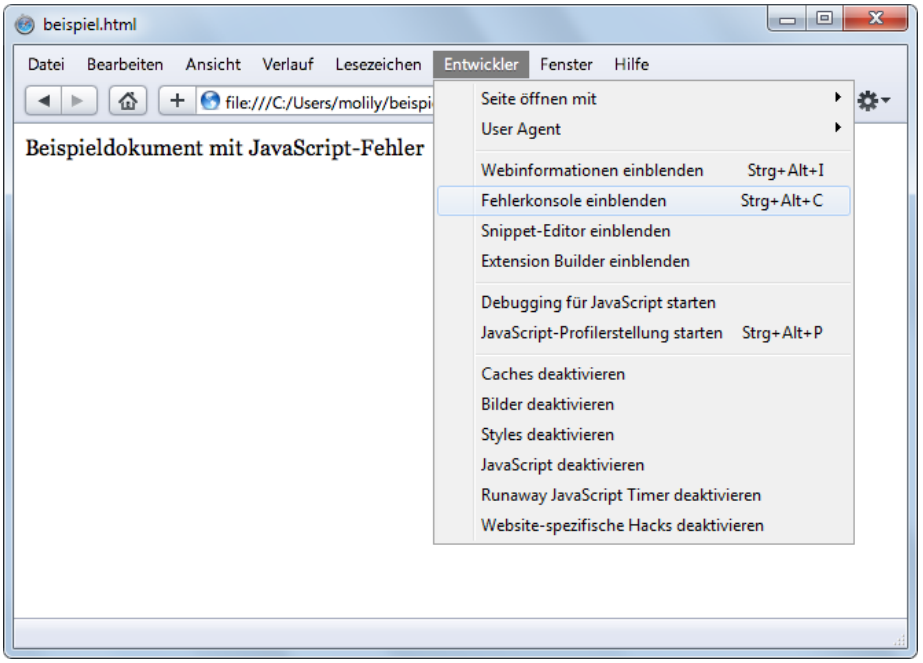
Im Safari unter Mac OS sieht dieser Dialog so aus:



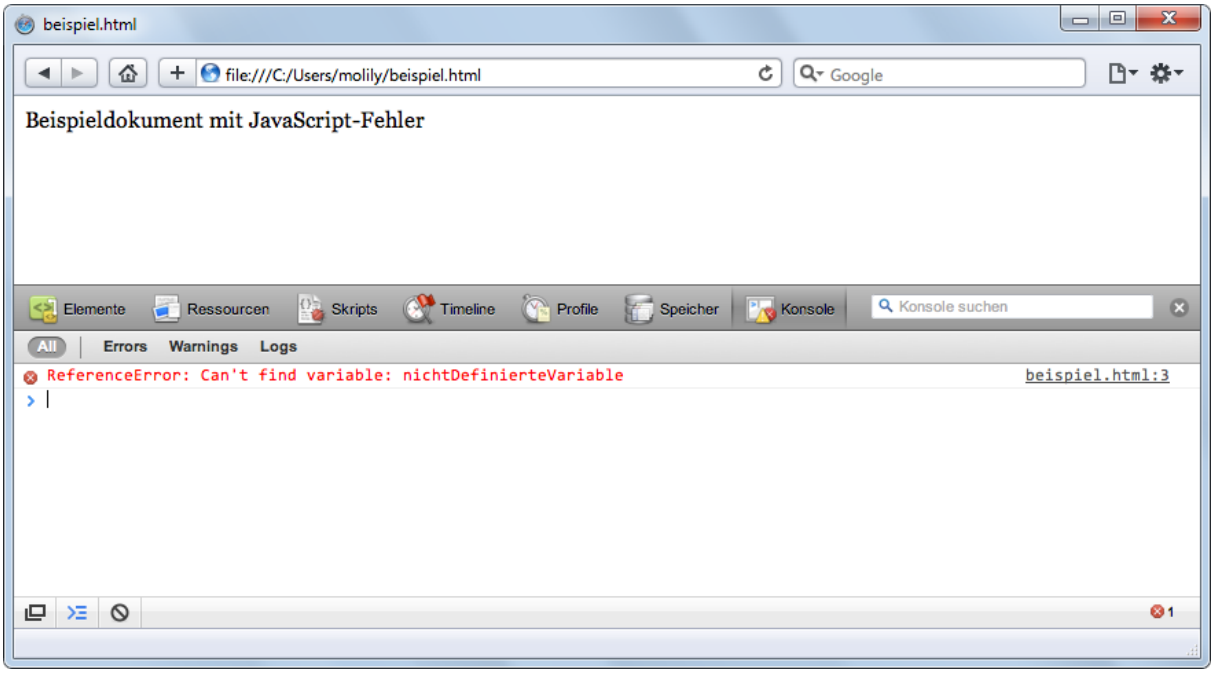
Dasselbe im Safari unter Windows:



Wenn sie nun die Einstellungen schließen, erscheint im Safari-Hauptfenster das Menü *Developer*, in dem Sie den Eintrag *Fehlerkonsole einblenden* finden:

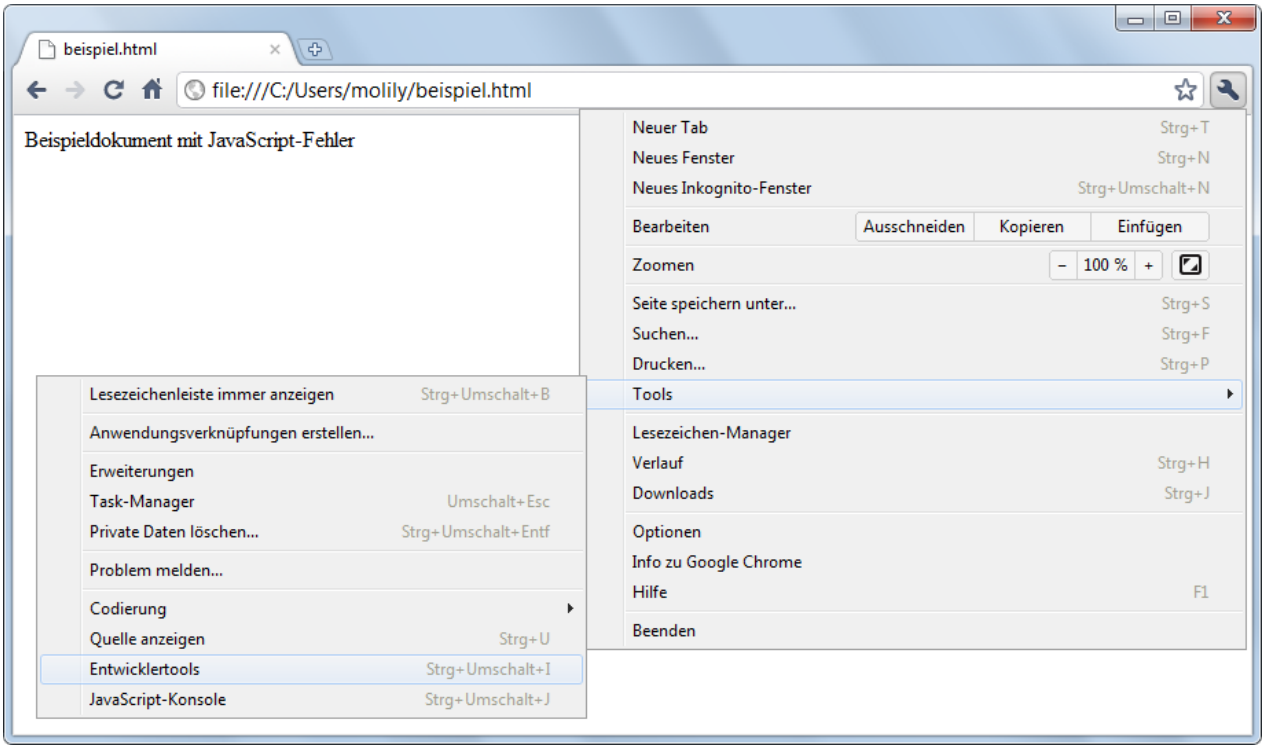


Diese Konsole ist Teil der Entwicklertools, welche Firebug-ähnlich typischerweise am unteren Rand des Browserfensters auftauchen. Der Beispielcode erzeugt darin folgende Fehlermeldung:

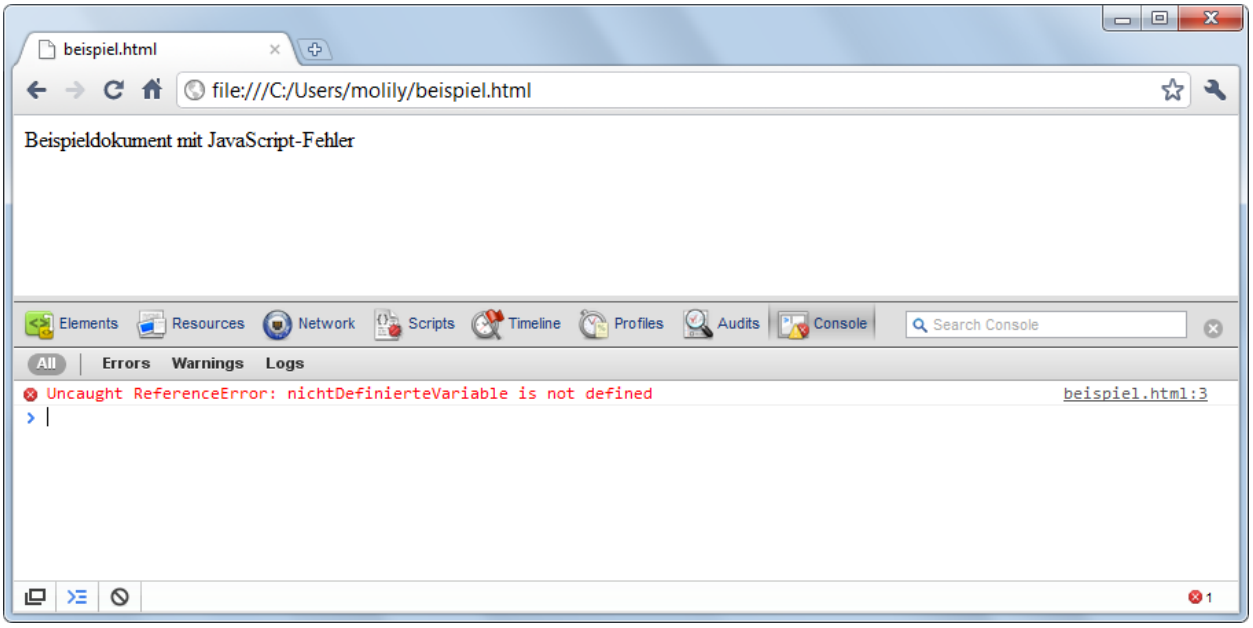


Mit einem Klick auf den Dateinamen können Sie in die Quelltext-Ansicht springen, um zur Code-Zeile zu gelangen, die den Fehler ausgelöst hat.

Im Chrome-Browser sieht die Fehlerkonsole sehr ähnlich aus. Sie öffnen die Konsole über *Tools* → *Entwicklertools* im Schraubenschlüssel-Menü.



Dadurch offenbart sich folgende Fehlermeldung in der Konsole:



Fehlersuche mit Kontrollausgaben

Hilfe, mein JavaScript funktioniert nicht! - Keine Panik. Wenn Sie bei der JavaScript-Programmierung ein paar Faustregeln befolgen, kommen Sie schnell voran und finden die Ursache.

Bei der JavaScript-Entwicklung sollten Sie sich eine methodische Herangehensweise angewöhnen: Werden Sie sich klar darüber, **was** sie genau vorhaben und **wie**, das heißt mit welchen Einzelschritten Sie das erreichen wollen. Wenn Sie sich diesen Programmablauf überlegt haben, recherchieren Sie die dazu nötigen JavaScript-Techniken. Suchen Sie in einer Objektreferenz die Objekte heraus, mit denen Sie arbeiten wollen. Lesen Sie die Beschreibungen aufmerksam durch, sodass Sie die Objekte korrekt verwenden können. Beachten Sie unter anderem den Typ von Eigenschaften und die Parameter, die Methoden entgegennehmen.

Wenn ein Script nicht wie erwartet funktioniert, sollten Sie zunächst in der *Fehlerkonsole* des Browsers schauen, ob in ihrem Script Fehler auftraten und der Browser die Ausführung deshalb gestoppt hat. Die Fehlerkonsole liefert in der Regel die Nummer der Codezeile in Ihrem Script, in der der Fehler aufgetreten ist. Die letztliche Ursache des Fehlers *kann*, muss aber *nicht* genau in dieser Zeile liegen! Die Ursache kann genauso eine Anweisung in einer anderen Zeile im Script sein, die vorher ausgeführt wurde, sodass die Voraussetzungen für die spätere Anweisung nicht gegeben sind.

Kontrollausgaben über die `alert`-Methode

Um Fehlern auf die Spur zu kommen, gehen Sie Ihr JavaScript-Programm Schritt für Schritt durch und notieren nach jeder relevanten Anweisung eine **Kontrollausgabe**. Dies geht am einfachsten mit der browserübergreifenden Methode `window.alert`, die ein Meldungsfenster erzeugt. Ein Beispiel:

```
var element = document.getElementById("navigation");
window.alert("Navigationselement: " + element);

var textknoten = document.createTextNode("Hallo!");
window.alert("Neuer Textknoten: " + textknoten);

element.appendChild(textknoten);
window.alert("Textknoten wurde angehängt.");
```

Wenn ein Fehler in Ihrem Script auftritt, dann bricht der Browser die Script-Ausführung ab, sodass die darauffolgende Kontrollausgabe nicht mehr ausgeführt wird. Somit können Sie einerseits den Fehler lokalisieren und andererseits gibt Ihnen die letzte Kontrollausgabe vor dem Abbruch eventuell Hinweise auf die Fehlerursache.

Um Fehler auf diese Weise eingrenzen zu können, sollten Sie die Anweisungen zwischen den Kontrollausgaben möglichst einfach halten - zumindest solange Ihr Script noch in der Entwicklungsphase ist. Die folgende Anweisung ist zusammengesetzt und kann an gleich mehreren Stellen zu einem Programmabbruch führen:

```
document.getElementsByTagName("h1")[0].style.color = "red";
```

Wenn in dieser Zeile ein JavaScript-Fehler auftritt, dann sollten Sie die Anweisung in ihre Teile aufspalten und Kontrollausgaben einfügen:

```
window.alert("Suche alle Überschriften im Dokument...");
var überschriftenListe = document.getElementsByTagName("h1");
window.alert("Anzahl der gefundenen h1-Überschriften: " + überschriftenListe.length);

var ersteÜberschrift = überschriftenListe[0];
window.alert("Erste h1-Überschrift: " + ersteÜberschrift);

window.alert("Setze Farbe der Überschrift...");
ersteÜberschrift.style.color = "red";
window.alert("Farbe gesetzt.");
```

Dieses Beispiel ist übertrieben, soll aber die Möglichkeiten von Kontrollausgaben veranschaulichen. Der Vorteil von `window.alert` ist, dass der Browser die Ausführung des Scriptes solange anhält, wie das Meldungsfenster geöffnet ist. Auf die Weise können Sie einem Script, das sonst ganz schnell ausgeführt wird, bei der Arbeit zuschauen und Fehler erkennen.

Der Nachteil von Kontrollausgaben mit `window.alert` ist, dass es JavaScript-Objekte in Strings umwandeln muss, um sie auszugeben. (Intern wird die Objektmethode `toString` aufgerufen.) Der String, der dabei herauskommt, gibt Ihnen nicht immer Aufschluss darüber, ob es sich um das erwartete Objekt handelt. Nur in einigen Browsern ist nachvollziehbar, um was für ein Objekt es sich handelt. Bei komplexen Objekten wie Arrays oder DOM-Knoten ist dies aber gewöhnungsbedürftig - mit der Zeit lernen Sie, wie sie solche Knoten am besten in Kontrollausgaben ansprechen.

Ablaufprotokolle mit Konsolen-Ausgaben (`console`-Objekt)

Vielseitiger als Ausgaben mittels `window.alert` ist die Nutzung der Schnittstelle zur JavaScript-Konsole des Browsers. Diese wurde von Firebug für den Firefox erfunden, wurde aber mittlerweile vom Internet Explorer, Safari und Chrome für ihre Fehlerkonsolen zumndest rudimentär übernommen:Dokumentation der Konsole-API von Firebug.

Das globale Objekt `window.console` bietet verschiedene Methoden an, mit denen Sie Statusmeldungen in die browsereigene Konsole schreiben können. In der Konsole haben Sie dann ein Log aller Meldungen Ihres Scriptes - ebenso wie etwaige Fehlermeldungen.

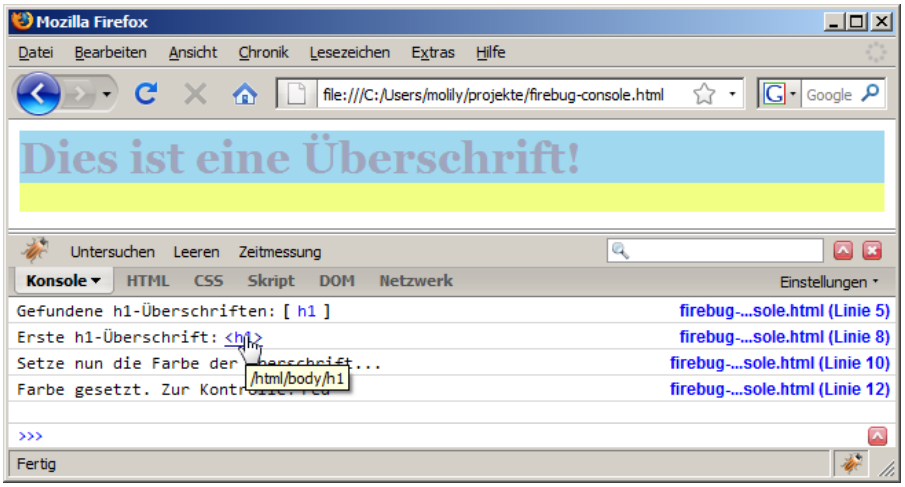
Das obige Beispiel könnte für Firebug mithilfe der Methode `console.debug` folgendermaßen umgesetzt werden:

```
var überschriftenListe = document.getElementsByTagName("h1");
console.debug("Gefundene h1-Überschriften:", überschriftenListe);

var ersteÜberschrift = überschriftenListe[0];
console.debug("Erste h1-Überschrift:", ersteÜberschrift);

console.debug("Setze nun die Farbe der Überschrift...");
ersteÜberschrift.style.color = "red";
console.debug("Farbe gesetzt. Zur Kontrolle:", ersteÜberschrift.style.color);
```

Ein Vorteil dieser Konsolenausgabe liegt darin, dass Firebug auch komplexe Objekte sinnfällig darstellt und so detaillierte Einblicke in die verwendeten Objekte und Werte erlaubt. Die Ausgaben in der Konsole sind interaktiv, d.h. sie können ein dort ausgegebenes Objekt anklicken und sich dessen Eigenschaften auflisten lassen.



Das Objekt `console` bietet viele weitere Methoden, um Meldungen in die Konsole zu schreiben. Diese können Sie der besagten Dokumentation entnehmen. Für den Anfang sollten Ihnen aber die Methoden `console.debug` und `console.log` ausreichen.

Auf diese Weise können Sie akribische **Ablaufprotokolle** Ihrer Scripte anlegen und haben die Objekte, mit denen Sie arbeiten, direkt im Blick. Dies macht Firebug zu einem unverzichtbaren Werkzeug. Mittlerweile bieten auch Entwicklerwerkzeuge für andere Browser ein `console`-Objekt mit den besagten Methoden an. Damit entsteht ein Quasi-Standard, mit dem hoffentlich irgendwann eine browserübergreifende JavaScript-Fehlersuche möglich sein wird.

Beachten Sie jedoch, dass Sie diese Methodenaufrufe nur während der Entwicklung ihrer Scripte nutzen können und am Ende wieder entfernen sollten - denn bei Ihren Webseiten-Besuchern wird das`console`-Objekt möglicherweise nicht zur Verfügung stehen. Ein Aufruf würde daher einen Fehler erzeugen und das Script abbrechen.

DOM-Inspektoren

Beim Einlesen eines HTML-Dokuments erzeugt der Browser aus dem HTML-Code eine Baumstruktur aus Elementen - diese Vorgang nennt sich Parsing. Die auf HTML aufbauenden Webtechniken CSS und JavaScript operieren auf dieser Grundlage. Den JavaScript-Zugriff auf diese interne Baumstruktur regelt das Document Object Model (DOM).

Zum Auslesen und Verändern des Dokuments bewegt sich ein Script im DOM-Baum. Um den Elementenbaum des aktuellen Dokuments zu veranschaulichen, bieten verschiedene Browser sogenannte **DOM-Inspektoren** an. Diese bilden die Knotenstruktur ab, die sie aus dem Dokument generiert haben, und erlauben eine Navigation darin. Sie können in der Baumdarstellung einen Elementknoten anklicken, um die Lage im Dokument, seine JavaScript-Eigenschaften und -Methoden, die wirkenden CSS-Formatierungen sowie die Dimensionen zu sehen.

Eine Möglichkeit, Informationen zu einem gewissen Element abzurufen, nennt sich *Element untersuchen* oder ähnlich. Dabei können Sie einen beliebigen Bereich im Dokument anklicken und im DOM-Inspektor wird automatisch das zugehörige Element fokussiert.

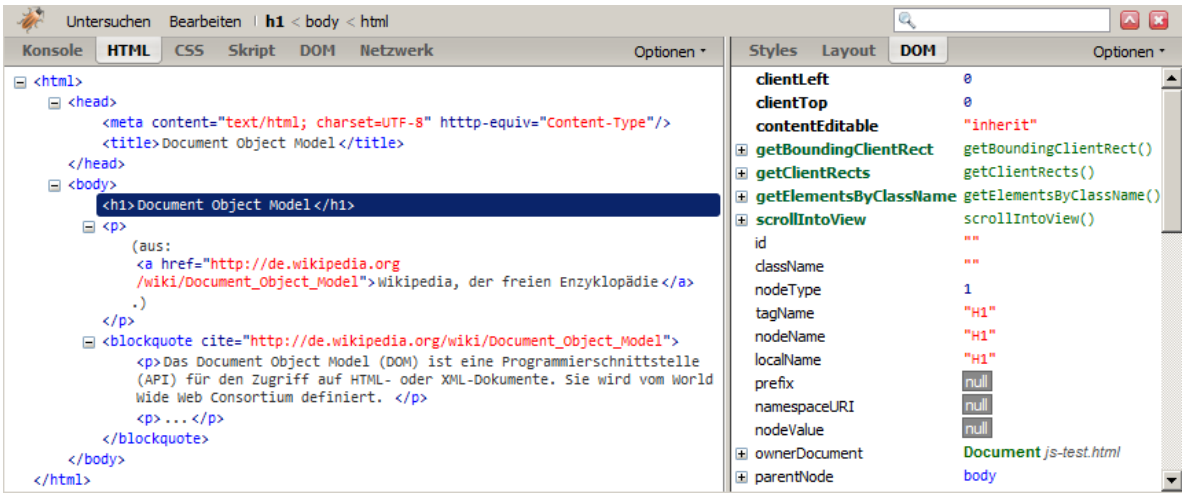
Nehmen wir folgendes Beispieldokument und schauen uns an, wie die verschiedenen DOM-Inspektoren den Elementenbaum wiedergeben:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Document Object Model</title>
</head>
<body>
<h1>Document Object Model</h1>
<p>(aus: <a href="http://de.wikipedia.org/wiki/Document_Object_Model">Wikipedia,
der freien Enzyklopädie</a>.)</p>
<blockquote cite="http://de.wikipedia.org/wiki/Document_Object_Model">
<p>Das Document Object Model (DOM) ist eine Programmierschnittstelle (API) für den Zugriff
auf HTML- oder XML-Dokumente. Sie wird vom World Wide Web Consortium definiert.
<p>...</p>
```

```
</blockquote>
</body>
</html>
```

DOM-Inspektor von Firebug

Zum DOM-Inspektor von Firebug gelangen Sie, indem Sie die Firebug-Leiste am unteren Fensterrand durch einen Klick auf das Käfer-Symbol aufklappen. Anschließend wählen Sie den Tab *HTML*. Auf der linken Seite findet sich der interaktive Elementbaum. Sie können die Verschachtelungen über die Plus-Symbole aufklappen, um die Kindelement eines Elements zu sehen. Um nähere Informationen zu einem Element zu bekommen, klicken Sie auf die blauen Elementnamen. In folgendem Screenshot wurde das *h1*-Element im Beispieldokument angeklickt:



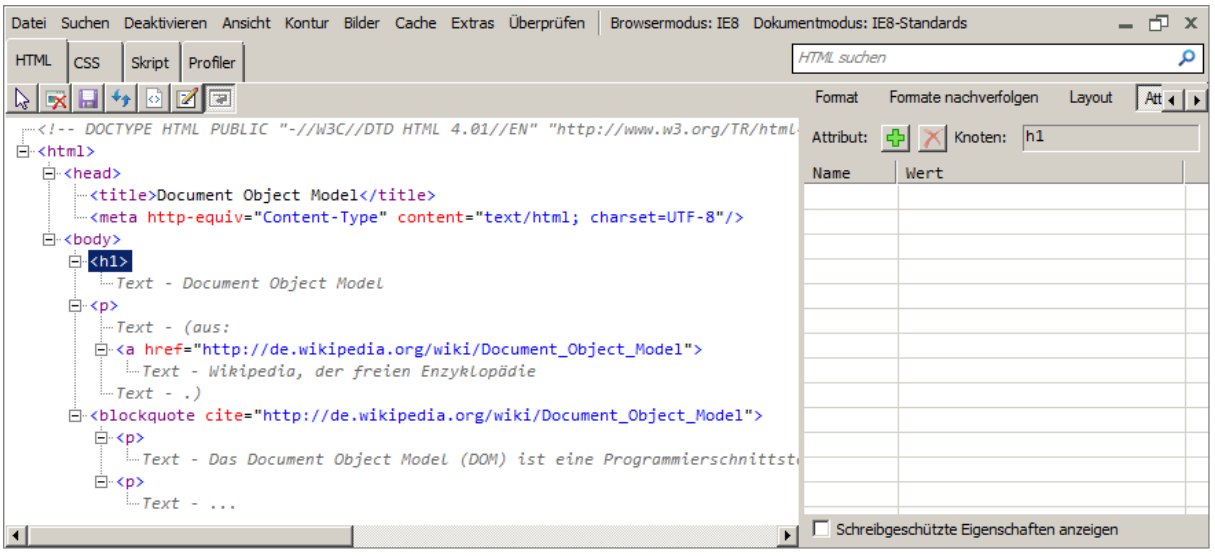
In der rechten Spalte finden Sie die drei Tabs *Styles*, *Layout* und *DOM*. Die ersten beiden sind in erster Linie bei der CSS-Entwicklung interessant, der dritte Tab insbesondere beim Schreiben von JavaScript. Darin sind alle Eigenschaften und Methoden des ausgewählten Elementobjekten aufgelistet. Viele dieser Eigenschaftswerte (z.B. `parentNode`) sind anklickbar und führen zur Ansicht des entsprechenden Knoten bzw. JavaScript-Objekts.

Um ein Element schnell zu finden, können Sie oben links in der Firebug-Leiste auf *Untersuchen*klicken. Nun können Sie im Dokument ein Element auswählen. Wenn Sie mit dem Mauszeiger durch das Dokument gehen, wird das Element, auf deren Fläche sich der Mauszeiger befindet, mit einem blauen Rahmen hervorgehoben. Außerdem wird das Element automatisch im DOM-Inspektor fokussiert. Klicken Sie ein Element an, um den Auswahl-Modus zu verlassen.

Alternativ zu dieser Vorgehensweise über die Schaltfläche *Untersuchen* können Sie ein Element im Dokument mit der rechten Maustaste anklicken und im Kontextmenü *Element untersuchen* wählen. Dies hat denselben Effekt.

Internet Explorer 8: DOM-Inspektor der Entwicklertools

...

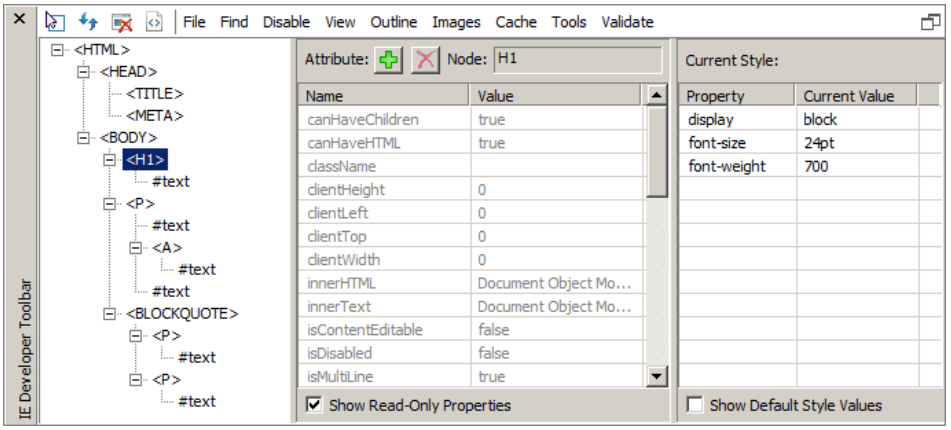


Internet Explorer 6 und 7: Developer Toolbar

Der Internet-Explorer-Versionen vor 8 verfügen über keinen eingebauten DOM-Inspektor. Es existiert allerdings ein offizieller Zusatz: Die Internet Explorer Developer Toolbar. Diese ist ein Vorgänger der Entwicklertools im Internet Explorer 8 und die Bedienung gestaltet sich weitgehend gleich.

Nachdem Sie den Zusatz installiert haben, können Sie die Toolbar aktivieren, indem Sie im Menü*Ansicht → Explorer-Leiste → IE Developer Toolbar* aktivieren. Gegebenenfalls ist das Menü im Internet Explorer 7 standardmäßig ausgeblendet - Sie können es kurzzeitig einblenden, indem Sie die Alt-Taste drücken. Die Toolbar erscheint am unteren Fensterrand und Sie können deren Höhe variieren.

Das Beispieldokument wird in der IE Developer Toolbar folgendermaßen veranschaulicht:



Die mittlere Spalte zeigt die HTML-Attribute und einige JavaScript-Eigenschaften des ausgewählten Elementobjektes an (hier wieder die h1-Überschrift). Die rechte Spalte listet CSS-Eigenschaften auf, die auf das gewählte Element wirken.

Um ein Element durch Anklicken zu untersuchen, klicken Sie das Icon oben links, auf dem ein Cursor und ein blaues Rechteck abgebildet sind. Damit befinden Sie sich im Element-Auswahlmodus und können den Mauszeiger über das Dokument bewegen und durch Klicken auswählen.

DOM-Inspektor von Opera Dragonfly

Den DOM-Inspektor von Opera Dragonfly erreichen Sie über das Menü *Extras → Weiteres → Opera Dragonfly*. Den DOM-Baum finden Sie im Tab *DOM*. Mittels Schaltflächen können Sie zwischen verschiedene Darstellungen wechseln und manche Knotentypen ausblenden. Im rechten Bereich können Sie über drei Tabs *Styles*, *Properties* und *Layout* auf die Elementeigenschaften zugreifen. Diese entsprechen funktional den Tabs *Styles*, *DOM* und *Layout* in Firebug.

Wie auch die anderen vorgestellten DOM-Inspektoren bietet Opera an, ein Element durch Klicken im Dokument auszuwählen. Auf der entsprechenden Schalftläche ist ein Cursor abgebildet.

...

Safari und Chrome Web Inspector

...

Debugger

Mithilfe von Kontrollausgaben können Sie den Ablauf eines Scriptes nachvollziehen und Variablen ausgeben, um deren Werte zu prüfen. Ein vielseitigeres und mächtigeres Werkzeug sind Debugger. Damit können Sie komfortabler die Funktionsweise eines Scriptes untersuchen, denn Sie müssen nicht nach jeder Anweisung eine Kontrollausgabe einfügen. Ein JavaScript-Debugger bietet im Allgemeinen folgende Möglichkeiten:

Sie können im Code sogenannte **Haltepunkte** (englisch *Breakpoints*) setzen. Die Ausführung des JavaScripts wird an dieser Stelle unterbrochen und der Browser öffnet den Debugger. Ausgehend von dieser Code-Zeile können Sie die folgenden Anweisungen nun Schritt für Schritt ausführen, Anweisungen überspringen und aus aufgerufenen Funktionen herausspringen. Bei dieser **schrittweisen**

Ausführungkönnen Sie überprüfen, welche Werte bestimmte Objekte und Variablen an dieser Stelle im Script haben. Eine Weiterentwicklung dieser **Überwachung von Variablenwerten** stellen sogenannte **Watch-Expressions** (englisch für Überwachungs-Ausdrücke) dar.

...

Wenn Sie viele Funktionen definieren, die sich gegenseitig aufrufen, hilft Ihnen ein Debugger, die Übersicht über das Aufrufen und Abarbeiten von Funktionen zu behalten. In der Einzelschritt-Ausführung haben Sie den sogenannten **Call Stack** (englisch für Aufruf-Stapel) im Blick. Das ist die Verschachtelung der Funktionen, die gerade abgearbeitet werden. Wenn beispeilsweise die Funktion a die Funktion baufruft und diese wiederum die Funktion c, so ist der Stapel a > b > c. Die Funktion c, die gerade ausgeführt wird, liegt sozusagen oben auf dem Stapel. Nach dessen Ausführung wird c vom Stapel genommen und in die Funktion b zurückgekehrt - und so weiter.

...

Debugging mit Firebug

Debugging mit den Internet Explorer Entwicklertools

Debugging mit Microsoft Visual Web Developer Express

Debugging von Opera Dragonfly

Debugging mit Safari und Chrome Web Inspector

Editoren und Entwicklungsumgebungen

Aptana Studio, Netbeans IDE, ...

JavaScript-Lints

- JSLint – The JavaScript Code Quality Tool
- JSHint – community-driven tool to detect errors and potential problems in JavaScript code

Code-Komprimierer und -Formatierer

Bei der JavaScript-Programmierung sollten Sie gewisse Konventionen zur Code-Formatierung einhalten. Das dient nicht nur der Lesbarkeit, Übersichtlichkeit und Klarheit des Quellcodes, sondern macht erst effektive Fehlersuche und Debugging möglich. Das beinhaltet grob gesagt eine Anweisung pro Zeile sowie eine Einrückung innerhalb von Blöcken (Funktionen, Kontrollstrukturen usw.). Wie das im Detail aussieht, ist Ihrem Geschmack überlassen.

Wenn das JavaScript jedoch im Browser ausgeführt wird, ...

- Google Closure Compiler
- Closure Compiler Service
- uglifyjs

JavaScript: Einbindung in HTML mit dem `script`-Element

Vorbemerkung

Von den vielen Möglichkeiten, JavaScript in HTML-Dokumente einzubetten, werden hier nur wenige gängige vorgestellt und empfohlen. Dieses Kapitel geht davon aus, dass HTML und JavaScript möglichst getrennt werden und sich JavaScripte eigenständig hinzuschalten. Die Hintergründe zu diesem Ansatz finden Sie im Kapitel Sinnvoller JavaScript-Einsatz.

Das script-Element

Zur Einbindung von JavaScript-Code in HTML-Dokument existiert in das HTML-Element `script`. Dieses darf sowohl im Kopf (`head`-Element) als auch im Körper (`body`-Element) eines HTML-Dokuments auftauchen. Es kann entweder direkt JavaScript-Code beinhalten, wie in diesem Beispiel:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Dokument mit integriertem JavaScript</title>
<script type="text/javascript">
window.alert("Hallo Welt!");
</script>
</head>
<body>
<h1>Dokument mit integriertem JavaScript</h1>
</body>
</html>
```

Oder es kann leer sein und auf eine externe Datei mit JavaScript-Code verweisen. Diese Nutzungsweise sollten Sie vorziehen und Ihre JavaScripte möglichst in separate Dateien auslagern.

Schreiben sie dazu Ihren JavaScript-Code in eine eigene Datei und speichern Sie sie mit der Dateiendung `.js` ab. Notieren Sie im Dokumentkopf ein `script`-Element, das den Browser auf die externe JavaScript-Datei hinweist. Dazu notieren Sie im `src`-Attribut die Adresse (URI), unter der das Script abrufbar ist. Vergessen Sie auch nicht das Attribut `type` mit dem festen Wert`text/javascript`. Dieses teilt dem Browser unmissverständlich mit, dass sie es sich bei ihrem Code um JavaScript handelt.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Dokument mit externem JavaScript</title>
<script type="text/javascript" src="script.js"></script>
</head>
<body>
<h1>Dokument mit externem JavaScript</h1>
</body>
</html>
```

In der Datei *script.js* können Sie nun JavaScript-Anweisungen, Funktionen, Objekte und so weiter notieren. Selbstverständlich können Sie mit dieser Methode mehrere Scripte einbinden: Verwenden Sie dazu einfach mehrere `script`-Elemente.

Beachten Sie, dass diese eingebundenen Dateien direkt JavaScript-Code enthalten müssen. HTML-Code darf darin nicht vorkommen – in JavaScript-Strings ist er natürlich noch erlaubt. Insbesondere ist es nicht nötig, den JavaScript-Code in der separaten Datei noch einmal in ein `script`-Element zu verpacken. Dies würde dazu führen, dass der Browser den JavaScript-Code nicht korrekt ausführt.

Ihre Scriptdateien können Sie – genauso wie Stylesheets, Grafiken usw. – auch in Unterverzeichnissen und sogar auf anderen Webservern unterbringen. Solange die angegebene URI korrekt ist, wird ein JavaScript-fähiger Browser sie beim Anzeigen des Dokuments herunterladen und ausführen.

Ausführung von script-Elementen

Mit dem `script`-Element können Sie sowohl Scripte im Dokumentkopf als auch im Dokumentkörper einbetten. Die Ausführung des Scriptcodes läuft nach gewissen Regeln ab, die wir im folgenden betrachten.

Wenn der Browser das HTML-Dokument vom Webserver empfängt, beginnt er sofort damit, den Quellcode zu verarbeiten und in eine interne Speicherstruktur, das Document Object Model (DOM) zu überführen. Das dafür zuständige Modul im Browser nennt sich **Parser** und der Verarbeitungsvorgang**Parsen**.

Sobald der Parser auf ein `script`-Element trifft, wird das Parsing des HTML-Dokuments angehalten und der JavaScript-Code innerhalb des `script`-Elements ausgeführt. Dasselbe gilt für externe JavaScript-Dateien: Der HTML-Parser stoppt, lädt die externe JavaScript-Datei vom Webserver, führt den JavaScript-Code aus und fährt erst dann mit der Verarbeitung des restlichen HTML-Quellcodes fort.

Diese Vorgehensweise, den JavaScript-Code direkt beim Einlesen des HTML-Dokuments auszuführen, hat folgende Konsequenzen:

Scripte haben Zugriff auf die Objekte vorher eingebundener Scripte

Die `script`-Elemente samt enthaltenem JavaScript-Code bzw. aus externen Dateien eingebundener JavaScript-Code werden in der Reihenfolge ausgeführt, in der sie im HTML-Quelltext notiert sind. Wenn Sie verschiedene Scripte haben, die aufeinander aufbauen, so müssen Sie sie nacheinander einbinden.

```
<script type="text/javascript" src="grundlagenscript.js"></script>
<script type="text/javascript" src="aufbauscript.js"></script>
<script type="text/javascript">
// Anwendung der Scripte
helferfunktion();
</script>
```

Das Beispiel bindet drei Scripte ein, die ersten beiden als externe Dateien, das dritte direkt im HTML-Code. Da der Browser die Scripte in der Reihenfolge ihrer Einbindung ausführt, können spätere Scripte die Objekte, Funktionen und Variablen nutzen, die die vorher eingebundenen Scripte definiert haben. Im Beispiel wird zuerst *grundlagenscript.js* eingebunden, heruntergeladen und ausgeführt. Das darauffolgende Script aus der Datei *aufbauscript.js* kann die darin notierten Funktionen nutzen. Schließlich kann das dritte Script eine Funktion nutzen, die in *aufbauscript.js* definiert wurde.

Externe Scripte verzögern den Aufbau des Dokuments

Dass der Webbrowser die eingebundenen Scripte nicht erst *nach*, sondern bereits *während* dem Einlesen des HTML-Codes ausführt, hat Vor- und Nachteile. Einerseits werden Scripte so schnell wie möglich ausgeführt und es ist garantiert, dass ein externes Script ausgeführt wird, bevor ein nachfolgendes internes Script abgearbeitet wird. Andererseits verlangsamt sich der Seitenaufbau, wenn große externe Scriptdateien vom Webserver heruntergeladen werden.

Dieser Nachteil kann dadurch umgangen werden, alle `script`-Elemente in der notwendigen Reihenfolge am Dokument-Ende zu platzieren anstatt wie üblich in den Dokumentkopf. Aus Gründen der kürzeren Ladezeit und des schnelleren Aufbau des Dokumentes wird dies immer öfter empfohlen. Es setzt allerdings eine bestimmte Arbeitsweise voraus. Im Abschnitt Ereignisbasierte Scripte werden wir eine Methode kennenlernen, bei die Scripte die Hauptarbeit erst verrichten, wenn das Dokument vollständig geladen wurde.

Scripte können während des Ladens das Dokument mit `document.write` ergänzen

Mit der Methode `document.write` kann ein Script schon während dem Laden das Dokument direkt beeinflussen und einige Weichen stellen. `document.write` nimmt HTML-Code in einem JavaScript-String entgegen und fügt diesen an der Stelle ins Dokument ein, an denen das zugehörige`script`-Element steht.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Zugriff auf das DOM während dem Parsen des Dokuments</title>
<script type="text/javascript">
document.write("<link rel='stylesheet' href='javascript.css'>");
</script>
</head>
<body>
<script type="text/javascript">
document.write("<p><a href='javascript:location.reload()'>" +
        "Seite mittels JavaScript neu laden<\/a><\/p>");
</script>
</body>
</html>
```

Das Beispiel enthält zwei Scripte mit `document.write`-Aufrufen. Diese schreiben HTML-Elemente ins Dokument, einmal ein Verweis auf ein Stylesheet und einmal ein Textabsatz mit einem JavaScript-Link.

`document.write` ist beim »Unobtrusive JavaScript« nur sehr selten sinnvoll. Inhalte, die nur bei aktiviertem JavaScript sichtbar sein sollen, da sie auf JavaScript-Funktionalität beruhen, sollten Sie ohne`document.write` dynamisch ins Dokument einfügen. Die dazu nötigen Techniken werden wir noch kennenlernen.

Der Anwendungsbereich von `document.write` wird oftmals missverstanden. Wir haben hier den einen von zwei möglichen Anwendungsfällen kennenlernt: Das Ergänzen eines Dokuments noch während der Browser den HTML-Code einliest.
Wenn `document.write` jedoch nach dem vollständigen Einlesen de HTML-Codes aufgerufen wird, hat die Methode einen ganz anderen Effekt und eignet sich nicht dazu, das vorhandene Dokument via JavaScript zu ändern.

Ein Script hat Zugriff auf die Elemente vor dem zugehörigen `script`-Element

Wie Sie vielleicht wissen, ist die häufigste Aufgabe von JavaScripten der Zugriff auf das Dokument über die DOM-Schnittstelle, die die Elemente und deren Textinhalte als Knotenobjekte zugänglich macht. Da ein Script mitten im Parsing-Prozess, also während des Einlesens des HTML-Dokuments ausgeführt wird, hat es zu diesem Zeitpunkt noch nicht Zugriff auf den gesamten DOM-Elementenbaum. Stattdessen kann es nur auf einen Teil-Baum zugreifen, nämlich auf die Elemente, die *vor* dem zugehörigen `script`-Element liegen und somit vom Parser bereits verarbeitet wurden.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Zugriff auf das DOM während dem Parsen des Dokuments</title>
<script type="text/javascript">
// Der Titel ist an dieser Stelle bereits verfügbar:
window.alert( document.title );
// Der Dokumentkörper allerdings noch nicht (ergibt null):
window.alert( document.body );
// Die Überschrift ebensowenig (ergibt null):
window.alert( document.getElementById("überschrift") );
</script>
</head>
<body>

<script type="text/javascript">
window.alert( document.title ); // OK
// Der Dokumentkörper ist erst an dieser Stelle verfügbar:
window.alert( document.body );
```

```
// Die Überschrift allerdings noch nicht (ergibt null):
window.alert( document.getElementById("überschrift") );
</script>

<h1 id="überschrift">Beispielüberschrift</h1>
<script type="text/javascript">
window.alert( document.title ); // OK
window.alert( document.body ); // OK
// Die Überschrift ist erst an dieser Stelle verfügbar:
window.alert( document.getElementById("überschrift") );
</script>

</body>
</html>
```

Das Beispiel enthält drei Scripte, die jeweils versuchen, auf den Dokument-Titel (`title`-Element), den Dokument-Körper (`body`-Element) und eine Überschrift (`h1`-Element) zuzugreifen. Je nachdem, an welcher Stelle sich das Script und das angesprochene Element befinden, ist der Zugriff auf das Element möglich oder nicht. Der Zugriff funktioniert erst dann, wenn das anzusprechende Element dem jeweiligen Script vorangeht und bereits geparst wurde. Das Element muss dazu noch nicht abgeschlossen sein. Im Beispiel kann ein Script im `body`-Element bereits auf das geöffnete, aber noch nicht geschlossene `body`-Element zugreifen. Das Script hat jedoch nur Zugriff auf die vorherigen Geschwisterelemente (im Beispiel das `h1`-Element).

Dies heißt nun nicht, dass sie zwangsläufig all ihre Scripte ans Dokumentende setzen müssen, damit ihr Script auf das gesamte Dokument zugreifen kann. Dieser Sachverhalt soll Ihnen nur die Ausgangssituation für ereignisbasierte Scripte schildern, die automatisch ihre Arbeit aufnehmen, sobald der gesamte HTML-Code geparst wurde und das Dokument fertig geladen ist.

Das `noscript`-Element

Das `noscript` ist als Gegenstück zu `script` gedacht: Damit lassen sich Alternativinhalte für Programme ins Dokument einfügen, die keine Scripte unterstützen. Browser, in denen JavaScripte deaktiviert oder nicht verfügbar ist, zeigen den Alternativinhalt an. Der Inhalt richtet sich auch an Programme wie Suchmaschinen-Robots, die das Dokument automatisiert verarbeiten, ohne die Scripte dabei zu beachten.

```
<noscript>
   <p>Dieser Absatz ist gehört zum Inhalt des Dokuments, ist aber
   im Browser nur zu sehen, wenn JavaScript deaktiviert oder nicht
   zur Verfügung steht.</p>
</noscript>
```

Der Sinn von `noscript` ist, die Informationen zugänglich zu machen, die sonst nur mithilfe des Scriptes zugänglich wären oder sogar durch das Script eingefügt werden. Diese Abwärtskompatibilität einer Website und die Zugänglichkeit aller Inhalte ohne JavaScript ist zwar erstrebenswert, allerdings zäumt »Unobtrusive JavaScript« das Pferd von vorne anstatt von hinten auf: Alle Informationen liegen bereits im Dokument und es ist auch ohne JavaScript gut bedienbar. Mittels JavaScript werden dann Zusatzfunktionen eingebaut, die die Bedienung und das Lesen der Inhalte vereinfachen und verbessern.

Im Unobtrusive JavaScript kommt dem `noscript`-Element daher keine Bedeutung zu. Von seiner Verwendung wird sogar abgeraten, da es dazu verleitet, JavaScript-Logik mit dem HTML-Code fest zu verschweißen, anstatt diese sauber zu trennen. Gestalten Sie Ihre Website so, dass ohne JavaScript möglichst alle Inhalte zugänglich sind und alle Funktionen zur Verfügung stehen. Ihre JavaScripte schalten sich dann hinzu und modifizieren das Dokument entsprechend. Anstatt also mittels `noscript`ein Element einzufügen, das nur ohne JavaScript relevant ist, sollten Sie dieses ganz normal ohne`noscript` im Dokument notieren. Falls es bei aktiviertem JavaScript nicht benötigt wird, dann können Sie es mittels JavaScript verändern oder ganz ausblenden.

Aus den besagten Gründen wird an dieser Stelle nicht näher auf `noscript` eingegangen – in den meisten Fällen werden Sie `noscript` nicht brauchen. Es gibt nur einige Spezialfälle, in denen`noscript` angemessen ist: Etwa wenn es sich bei der Website um eine reine JavaScript-Webanwendung handelt, die (noch) keine Alternativversion anbietet. Dann können Sie mit `noscript`einen Hinweis darauf hinterlegen, dass die Site einen JavaScript-fähigen Browser zwingend voraussetzt.

JavaScript: Grundlagen zur Ereignisverarbeitung

Ereignisbasierte Scripte

Der Abschnitt über die Verarbeitung von Scripten hat Ihnen gezeigt, dass der Browser Scripte üblicherweise in dem Moment ausführt, in dem er den Code eines HTML-Dokuments herunterlädt, parst und auf ein `script`-Element trifft.

Der Schicksal von JavaScript ist aber nicht, bloß in diesem kurzen Moment des Ladens des HTML-Dokuments ausgeführt zu werden und dann für immer zur Ruhe zu kommen. Die meisten JavaScripte sollen Interaktivität bieten. Der Schlüssel dazu ist, das haben wir bereits in den Grundkonzeptenkennengelernt, die **Überwachung und Behandlung von Ereignissen** (auch **Event-Handling** genannt).

Moderne Scripte durchlaufen deshalb verschiedene **Phasen**:

Phase Eins: Das Dokument wird empfangen und geparst

Dabei wird das JavaScript erstmals ausgeführt. Objekte und Funktionen werden dabei definiert, sodass sie für die spätere Nutzung zur Verfügung stehen. – Nicht alle notierten Funktionen werden dabei bereits aufgerufen. – Zu diesem Zeitpunkt hat das Script noch keinen vollständigen Zugriff auf das Dokument.

Phase Zwei: Das Dokument ist fertig geladen

Der vollständige Zugriff auf das Dokument über das DOM ist erst jetzt möglich. Nun wird ein Teil des Scripts aktiv, um dem bisher statischen Dokument JavaScript-Interaktivität hinzuzufügen: Das Script spricht vorhandene Elementknoten an und fügt ihnen sogenannte **Event-Handler** hinzu. Das Script kann aber auch den Inhalt oder die Darstellung von bestehenden Elementen verändern und dem Dokument neue Elemente hinzufügen (auch **DOM-Manipulation** genannt).

Phase Drei: Der Anwender bedient das Dokument und das Script reagiert darauf

Wenn die überwachten Ereignisse an den entsprechenden Elementen im Dokument passieren, so werden gewisse andere Teiles des Scripts aktiv, denn die entsprechenden **Handler-Funktionen** werden ausgeführt.

Resultierende Script-Struktur

Dieser chronologische Ablauf gibt die Struktur der meisten Scripte vor:

- Im Code werden vor allem mehrere Funktionen definiert, die später als *Handler* Ereignisse verarbeiten werden.
- Es gibt mindestens eine Hauptfunktion, die ausgeführt wird, wenn der Browser das Dokument fertig geladen hat, sodass Scripte darauf zugreifen können.

Das erste und enorm wichtige Ereignis, mit dem wir uns beschäftigen müssen, ist daher das`load`**-Ereignis**. Es passiert aus JavaScript-Sicht im Fenster beim `window`-Objekt. Wenn dieses Ereignis eintritt, wird die zweite Phase aktiv. Dem JavaScript steht der gesamte DOM-Baum zur Verfügung, einzelne Elemente werden angesprochen und es werden Event-Handler registriert. Diese

elementar wichtige Vorgehensweise bei der JavaScript-Programmierung wird uns nun beschäftigen.

Traditionelles Event-Handling

Die Anweisung, die die Überwachung eines Ereignisses an einem Element startet, nennt man das*Registrieren* von Event-Handlern. Im Folgenden wird es um die einfachste und älteste Methode gehen, um Event-Handler zu registrieren.

In den Grundkonzepten haben wir die typischen Bestandteile der Ereignis-Überwachung kennengelernt: Erstens ein Elementobjekt, zweitens den Ereignistyp (z.B. `click`) und drittens eine Handler-Funktion. Diese drei Bestandteile finden wir in dem Aufbau der JavaScript-Anweisung wieder. Das allgemeine Schema lautet allgemein:

```
element.onevent = handlerfunktion;
```

- `element` steht für ein JavaScript-Objekt aus dem Dokument, üblicherweise ein Elementknoten. Es kommen auch besondere Objekte wie `window` und `document` in Frage.
- `onevent` ist eine Objekteigenschaft, die mit der Vorsilbe `on` beginnt, auf die der Ereignistyp folgt.*on* ist die englische Präposition für *bei*. Zum Beispiel *onclick* bedeutet soviel wie *beim Klicken*.
- `handlerfunktion` ist der Name einer Funktion. Genauer gesagt steht an dieser Stelle ein beliebiger Ausdruck, der ein Funktionsobjekt ergibt: JavaScript wird diesen Ausdruck auflösen und das Ergebnis als Handler-Funktion verwenden.

  Insgesamt hat die Anweisung die Form »Führe bei *diesem Element* beim Eintreten *dieses Ereignisses diese Funktion* aus.«

Der obige Pseudocode soll nur das allgemeine Schema illustrieren. Es gibt natürlich kein Ereignis namens `event`, und `onevent` ist lediglich ein Platzhalter für alle möglichen Eigenschaften, darunter`onclick`, `onmouseover`, `onkeypress` und so weiter.

Betrachten wir ein konkretes Beispiel. Wir wollen nach dem erfolgreichen Laden des Dokuments eine JavaScript-Funktion ausführen. Dazu haben wir bereits das `load`-Ereignis kennengelernt, dass beim`window`-Objekt passiert. Angenommen, wir haben eine Funktion namens `start` definiert, die als Event-Handler dienen wird:

```
function start () {
    window.alert("Dokument erfolgreich geladen! Wir können nun über das DOM darauf zugreifen.");
}
```

Gemäß dem obigen Schema starten wir folgendermaßen das Event-Handling:

```
window.onload = start;
```

Und schon wird die gezeigte Funktion beim erfolgreichen Laden des Dokuments ausgeführt.

Sie werden sich sicher fragen, wie Ereignis-Verarbeitung auf JavaScript-Ebene funktioniert. Dazu schauen wir uns den Aufbau der besagten Anweisungen an: Wir haben dort eine einfache Wertzuweisung (erkennbar durch das `=`), die einer Objekteigenschaft (`window.onload` auf der linken Seite) einen Wert (`starte` auf der rechten Seite) zuweist. Nach dieser Zuweisung ist die Funktion in der Objekteigenschaft gespeichert. Dies funktioniert, weil Funktionen in JavaScripte auch nur Objekte sind, auf die beliebig viele Variablen und Eigenschaften verweisen können.

Passiert nun ein Ereignis am Objekt `window`, sucht der JavaScript-Interpreter nach einer Objekteigenschaft, die den Namen `on` gefolgt vom Ereignistyp trägt (im Beispiel `onload`). Wenn diese Eigenschaft eine Funktion beinhaltet, führt er diese aus. Das ist erst einmal alles – aber enorm wichtig zum Verständnis des Event-Handlings.

Wie Sie später erfahren werden, ist die oben vorgestellte Methode im Grunde überholt (sieheNachteile und Alternativen). Dieses *traditionelle* Event-Handling ist aber immer noch der Ausgangspunkt jeder JavaScript-Programmierung. Sie sollten sich dieses Schema und dessen Funktionsweise genau einprägen.

Beispiel für traditionelles Event-Handling

Mit dem Wissen über Ereignis-Überwachung und das `load`-Ereignis können wir ein Dokument mitsamt eines Scriptes schreiben, das die beschriebenen drei Phasen illustriert.

Dazu starten wir mit folgendem einfachen Dokument:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Dokument mit JavaScript</title>
<script type="text/javascript"> … </script>
</head>
<body>

<p id="interaktiv">Dies ist ein einfacher Textabsatz, aber mithilfe von JavaScript können wir
ihn interaktiv gestalten. Klicken Sie diesen Absatz doch einfach mal mit der Maus an!</p>

</body>
</html>
```

Dem `p`-Element mit der ID `interaktiv` soll nun per JavaScript ein Event-Handler zugewiesen werden. Ziel ist es, dass eine bestimmte JavaScript-Funktion aufgerufen wird, immer wenn der Anwender auf die Fläche des Element klickt. Das Ereignis, das bei einem Mausklick ausgelöst wird, heißt sinnigerweise `click`.

Unser Script läuft in drei Schritten ab:

1. **Warten, bis das Dokument vollständig geladen ist:** Starte die Überwachung des `load`-Ereignisses und führe eine Startfunktion aus, sobald das Ereignis passiert.
2. **Einrichtung der Event-Handler:** Die besagte Startfunktion spricht den Textabsatz an und registriert einen Event-Handler für das `click`-Ereignis.
3. **Ereignis-Verarbeitung:** Die Handler-Funktion, die beim Klick auf den Textabsatz ausgeführt wird.

Schritt 1 ist mit der Anweisung erledigt, die wir bereits oben kennengelernt haben:

```
window.onload = start;
```

Natürlich können wir der Startfunktion auch einen anderen Namen als `start` geben. Üblich ist z.B.`init`.

Die Startfunktion für Schritt 2 könnte so aussehen:

```
function start () {
   document.getElementById("interaktiv").onclick = klickverarbeitung;
}
```

Was zunächst kompliziert aussieht, ist nichts anderes als die Anwendung des bekannten Schemas`element.onevent = handlerfunktion;`.

Zur Einrichtung des Event-Handler greifen wir über das DOM auf das Dokument zu. Dies ist in der Funktion `start` möglich, denn sie wird beim Eintreten des `load`-Ereignisses ausgeführt.

Damit der Zugriff auf das gewünschte Element so einfach möglich ist, haben wir einen »Angriffspunkt« für das Script geschaffen, indem wir dem `p`-Element eine ID zugewiesen haben. Eine solche Auszeichnung über IDs und Klassen (`class`-Attribute) spielen eine wichtige Rolle, um Angriffspunkte für Stylesheets und Scripte zu bieten.

Mit der DOM-Methode `document.getElementById` (zu deutsch: *gib mir das Element anhand der folgenden ID*) können wir das Element mit der bekannten ID ansprechen. Der Aufruf`document.getElementById("interaktiv")` gibt uns das Objekt zurück, das das `p`-Element repräsentiert.

Wir arbeiten direkt mit diesem Rückgabewert weiter und weisen dem Elementobjekt nun einen Event-Handler zu. Die Objekteigenschaft lautet `onclick`, denn es geht um das `click`-Ereignis. Die auszuführende Handler-Funktion lautet `klickverarbeitung`, dieser Name ist willkürlich gewählt.

Das ist schon alles und damit kommen wir zur Definition der besagten Funktion`klickverarbeitung`:

```
function klickverarbeitung () {
   document.getElementById("interaktiv").innerHTML += " Huhu, das ist von Javascript eingefügter Text.";
}
```

Was darin passiert, müssen Sie noch nicht bis ins Detail verstehen. Wie Sie sehen können, wird darin ebenfalls mittels `document.getElementById` das angeklickte `p`-Element angesprochen. Erneut wird eine Eigenschaft gesetzt, diesmal `innerHTML`. An den bestehenden Wert wird mit dem Operator`+=` ein String angehängt. Wenn Sie das Beispiel im Browser ausführen und auf das Element klicken, ändert sich der Text des Elements.

Zusammengefasst sieht das Beispiel mit eingebettetem JavaScript so aus:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Beispiel für traditionelles Event-Handling</title>
<script type="text/javascript">

window.onload = start;

function start () {
   document.getElementById("interaktiv").onclick = klickverarbeitung;
}

function klickverarbeitung () {
   document.getElementById("interaktiv").innerHTML += " Huhu, das ist von Javascript eingefügter Text.";
}

</script>
</head>
<body>

<p id="interaktiv">Dies ist ein einfacher Textabsatz, aber mithilfe von JavaScript können wir ihn
interaktiv gestalten. Klicken Sie diesen Absatz doch einfach mal mit der Maus an!</p>

</body>
</html>
```

Den JavaScript-Code können wir später natürlich in eine externe Datei auslagern.

So einfach und nutzlos dieses kleine Beispiel aussieht: Wenn Sie das dreischrittige Schema verstanden haben, beherrschen Sie einen Großteil der JavaScript-Programmierung und wissen, wie Scripte üblicherweise strukturiert werden und schließlich ausgeführt werden.

Event-Überwachung beenden

Wenn Sie einmal einen Event-Handler bei einem Element registriert haben, wird die Handler-Funktion künftig bei jedem Eintreten des Ereignisses ausgeführt – zumindest solange das Dokument im Browser dargestellt wird und es nicht neu geladen wird. Es ist möglich, die Event-Überwachung wieder mittels JavaScript zu beenden.

Wie beschrieben besteht das traditionelle Event-Handling schlicht darin, dass eine Objekteigenschaft (z.B. `onclick`) durch eine Wertzuweisung mit einer Funktion gefüllt wird. Um das Registrieren des Handlers rückgängig zu machen, beschreiben wir erneut diese Objekteigenschaft. Allerdings weisen wir ihr keine Funktion zu, sondern einen anderen Wert. Dazu bietet sich beispielsweise der

spezielle Wert `null` an, der soviel wie »absichtlich leer« bedeutet.

Das Schema zum Löschen des Event-Handlers lautet demnach:

```
element.onevent = null;
```

Wenn wir im obigen Beispiel die Überwachung des Klick-Ereignisses beim `p`-Element wieder beenden wollen, können wir entsprechend notieren:

```
document.getElementById("interaktiv").onclick = null;
```

Häufiger Fehler: Handler-Funktion direkt aufrufen

Ein häufiger Fehler beim Registrierens eines Event-Handlers sieht folgendermaßen aus:

```
element.onevent = handlerfunktion();   // Fehler!
```

Oft steckt hinter dieser Schreibweise der Wunsch, der Handler-Funktion noch Parameter mitzugeben, damit darin gewissen Daten zur Verfügung stehen:

```
element.onevent = handlerfunktion(parameter);   // Fehler!
```

Sie müssen sich die Funktionsweise des traditionellen Event-Handlings noch einmal durch den Kopf gehen lassen, um zu verstehen, warum diese Anweisungen nicht den gewünschten Zweck erfüllen. Beim korrekten Schema `element.onevent = handlerfunktion;` wird eine Funktion, genauer gesagt ein Funktionsobjekt, in einer Eigenschaft des Elementobjektes gespeichert.

Das ist beim obigen fehlerhaften Code nicht der Fall. Anstatt auf das Eintreten des Ereignisses zu warten, wird die Handler-Funktion **sofort ausgeführt**. Dafür verantwortlich sind die Klammern `()` hinter dem Funktionsnamen – diese Klammern sind nämlich der JavaScript-Operator zum Aufruf von Funktionen.

Das Erste, was der JavaScript-Interpreter beim Verarbeiten dieser Zeile macht, ist der Aufruf der Funktion. Deren *Rückgabewert* wird schließlich in der `onevent`-Eigenschaft gespeichert. In den meisten Fällen hat die Handler-Funktion keinen Rückgabewert, was dem Wert `undefined` entspricht, oder sie gibt `false` zurück, sodass schlicht diese Werte in die Eigenschaft geschrieben werden. Wir wollen die Funktion aber nicht direkt aufrufen, sondern bloß das Funktionsobjekt ansprechen, um es in die Eigenschaft zu kopieren. Daher dürfen an dieser Stelle keine Klammern hinter dem Namen notiert werden.

Der Wunsch, der Handler-Funktion gewisse Daten als Parameter zu übergeben, ist verständlich. Die obige fehlerhafte Schreibweise vermag dies aber nicht zu leisten. Leider ist diese Aufgabenstellung auch nicht so einfach lösbar: Das altbekannte Schema `element.onevent = handlerfunktion;` muss eingehalten werden. Der Funktionsaufruf, der die Parameter übergibt, wird in einer zusätzlichen Funktion untergebracht (gekapselt). Schematisch:

```
function helferfunktion (parameter) {
   /* Arbeite mit dem Parameter und verarbeite das Ereignis */
}
function handlerfunktion () {
   helferfunktion("Parameter");
}
element.onevent = handlerfunktion;
```

Das konkrete Beispiel aus dem vorigen Abschnitt können wir so anpassen, dass in der Handler-Funktion bloß eine andere Hilfsfunktion mit Parametern ausgeführt wird:

```
window.onload = start;
```

```
function start () {
   document.getElementById("interaktiv").onclick = klickverarbeitung;
}

function klickverarbeitung () {
   textHinzufügen(document.getElementById("interaktiv"), "Huhu, das ist von Javascript eingefügter Text.");
}

function textHinzufügen (element, neuerText) {
   element.innerHTML += neuerText;
}
```

In der Handler-Funktion `klickverarbeitung` wird die neue Funktion `textHinzufügen` mit Parametern aufgerufen. Diese wurde verallgemeinert und ist wiederverwendbar: Sie nimmt zwei Parameter an, einmal ein Elementobjekt und einmal einen String. Die Funktion hängt sie den angegebenen Text in das angegebene Element ein.

Eingebettete Event-Handler-Attribute

Wir haben kennengelernt, wie wir externe JavaScripte einbinden und darin auf »traditionelle« Weise Event-Handler registrieren können. Der Vorteil davon ist, dass wir HTML- und JavaScript-Code und damit das Dokument und das JavaScript-Verhalten trennen können.

Wann immer es möglich ist, sollten Sie diese Vorgehensweise des »Unobtrusive JavaScript« wählen. Es soll aber nicht verschwiegen werden, dass es auch möglich ist, JavaScript direkt im HTML-Code unterzubringen und damit auf Ereignisse zu reagieren.

Zu diesem Zweck besitzen fast alle HTML-Elemente entsprechende Attribute, in die Sie den auszuführenden JavaScript-Code direkt hineinschreiben können. In diesem Code können Sie natürlich auch eigene Funktionen aufrufen, die sie in einem `script`-Element oder einer externen JavaScript-Datei definiert haben. Die Attribute sind genauso benannt wie die entsprechenden JavaScript-Eigenschaften: Die Vorsilbe `on` gefolgt vom Ereignistyp (z.B. `click`). Das Schema lautet dementsprechend:

```
<element onevent="JavaScript-Anweisungen">
```

Ein konkretes Beispiel:

```
<p onclick="window.alert('Absatz wurde geklickt!');">Klicken Sie diesen Textabsatz an!</p>
```

Hier enthält das Attribut die JavaScript-Anweisung `window.alert('Absatz wurde geklickt!');`, also einen Aufruf der Funktion `window.alert`. Sie können mehrere Anweisungen in einer Zeile notieren, indem Sie sie wie üblich mit einem Semikolon trennen. Zum Beispiel Funktionsaufrufe:

```
<p onclick="funktion1(); funktion2();">Klicken Sie diesen Textabsatz an!</p>
```

Wie sie sehen, wird es hier schon unübersichtlich. Sie müssen Ihren Code in eine Zeile quetschen, damit die Browser das Attribut korrekt verarbeiten.

Es gibt viele gute Gründe, HTML und JavaScript möglichst zu trennen und auf solches *Inline-JavaScript* zu verzichten. Natürlich hat diese Grundregel berechtigte Ausnahmen. Als Anfänger sollten sie sich jedoch mit der Trennung sowie dem Registrieren von Event-Handlern mittels JavaScript vertraut machen, wie es in den vorigen Abschnitten erläutert wurde. Wenn Ihre Scripte komplexer werden, werden Sie vielleicht vereinzelt auf Event-Handler-Attribute zurückgreifen, aber der Großteil sollte ohne sie funktionieren.

Die Verwendung von solchen Event-Handler-Attrbuten bringt viele Eigenheiten und Nachteile mit sich, auf die an dieser Stelle nicht weiter eingegangen wird.

(Zugriff auf this u.d. Event-Objekt, Mehrere Anweisungen und Whitespace, Stringbegrenzung/Anführungszeichen, geänderte Scope-Chain, siehe Forumsposting)

Häufiger Fehler: Auszuführenden Code als String zuweisen

Nachdem wir Inline-JavaScript angeschnitten haben, sei auf einen weiteren häufigen Fehler beim traditionellen Event-Handling hingewiesen. Manche übertragen ihr Wissen über Event-Handler-Attribute aus HTML auf das das Registrieren von Event-Handlern in JavaScript. Sie versuchen z.B. folgendes:

```
element.onclick = "window.alert('Element wurde geklickt!');"
```

Oder gleichwertig mithilfe der DOM-Methode `setAttribute`:

```
element.setAttribute("onclick", "window.alert('Element wurde geklickt!');");
```

Sprich, sie behandeln die Eigenschaft `onclick` und dergleichen wie Attribute unter vielen. Für die meisten anderen Attribute gilt das auch. Ein Beispiel:

```
<p><a id="link" href="http://de.selfhtml.org/"
   title="Deutschsprachige Anleitung zum Erstellen von Webseiten">SELFHTML</a></p>
<script type="text/javascript">
var element = document.getElementById("link");
element.title = "Die freie Enzyklopädie";
element.href = "http://de.wikipedia.org/";
element.firstChild.nodeValue = "Wikipedia";
</script>
```

Das Script spricht ein Link-Element über seine ID an und ändert dessen Attribute `title` und `href`sowie schließlich dessen Textinhalt. Das Beispiel illustriert, dass sich die Zuweisungen der Attributwerte im HTML und im JavaScript stark ähneln. Die neuen Attributwerte werden im JavaScript einfach als Strings notiert.

Diese Vorgehensweise ist beim Setzen von Event-Handler-Attributen über JavaScript nicht völlig falsch. *Theoretisch* haben folgende Schreibweisen denselben Effekt:

```
// Methode 1: Traditionelles Event-Handling
function handlerfunktion () {
   window.alert("Hallo Welt!");
}
element.onevent = handlerfunktion;

// Methode 2: Auszuführenden Code als als String zuweisen
// (Achtung, nicht browserübergreifend!)
element.setAttribute("onevent", "window.alert('Hallo Welt!');");
```

Ihnen mag die die zweite Schreibweise in vielen Fällen einfacher und kürzer erscheinen. Doch zum einen hat sie das Problem, dass sie in der Praxis längst nicht so etabliert ist wie die traditionelle: Der Internet Explorer einschließlich der neuesten Version 8 unterstützt diese Schreibweise noch nicht.

Davon abgesehen hat es Nachteile, JavaScript-Code nicht in Funktionen zu ordnen, sondern in Strings zu verpacken. Der Code wird unübersichtlicher und Fehler sind schwieriger zu finden. Sie sollten daher möglichst das traditionelle Schema vorziehen.

JavaScript: Arbeiten mit dem Event-Objekt

Zugriff auf das Event-Objekt

Durch das Registrieren von Event-Handlern wird die angegebene Funktion immer dann ausgeführt, wenn das jeweilige Ereignis beim jeweiligen Element eintritt. In dieser Handler-Funktion ist es meistens nötig, auf die näheren Umstände des Ereignisses zu reagieren. Beispielsweise sind bei einem Mausklick die Koordinaten des Mauszeigers interessant oder bei einem Tastendruck die gedrückte Taste.

All diese Informationen sind in JavaScript beim **Event-Objekt** gespeichert. Dieses Objekt repräsentiert das individuelle Ereignis, das der Handler gerade verarbeitet. Es bietet zahlreiche Eigenschaften mit Informationen zum Ereignis und einige Methoden, um das Verhalten des Ereignisses zu steuern. Wenn Sie bei der Ereignisverarbeitung diese Daten benötigen, ist der Zugriff auf das Event-Objekt die erste Aufgabe in der Handler-Funktion.

In den meisten Browsern gestaltet sich dieser Zugriff einfach: Das Event-Objekt wird der Handlerfunktion automatisch als erster Parameter übergeben. Sie muss dieses nur noch entgegen nehmen, der Parametername ist dabei frei wählbar. Üblicherweise wird ein Kurzname wie `e` oder `ev`verwendet. Für das folgende Beispiel wählen wir den sprechenden Namen `eventObjekt`:

```
function handlerfunktion (eventObjekt) {
   window.alert("Es ist ein Ereignis vom Typ " + eventObjekt.type + " passiert.");
   // Fehler im Internet Explorer < 9!
}
```

Diesen Zugriff auf das Event-Objekt unterstützen alle relevanten Browser. Lediglich ältere Internet Explorer vor der Version 9 unterstützen diese Technik nicht. Für diese Browserversionen ist eine Sonderlösung notwendig. Diese Internet Explorer übergeben das Event-Objekt nicht als Parameter an die Handlerfunktion, sondern stellen es unter dem globalen Objekt `window.event` zur Verfügung. Auch wenn es den Anschein hat: Dort ist das Event-Objekt nicht dauerhaft gespeichert, sondern`window.event` verweist lediglich auf das jeweilige Event-Objekt des Ereignisses, das gerade im jeweiligen Handler verarbeitet wird.

Um browserübergreifend auf das Event-Objekt zuzugreifen, ist also eine Vereinheitlichung notwendig. Diese ist recht einfach: Wir prüfen, ob der Funktion ein Parameter übergeben wurde und somit die lokale Variable `eventObjekt` einen Wert hat. Falls dies zutrifft, nehmen wir diesen Parameter als Event-Objekt. Andernfalls speichern wir in der bisher leeren Variable eine Referenz auf`window.event`.

```
function handlerfunktion (eventObjekt) {
   // Vereinheitlichung:
   if (!eventObjekt) {
      // Korrektur für den Internet Explorer < 9
      eventObjekt = window.event;
   }

   // Browserübergreifender Zugriff:
   window.alert("Es ist ein Ereignis vom Typ " + eventObjekt.type + " passiert.");
}
```

Nach der Vereinheitlichung steht das Event-Objekt browserübergreifend in einer Variable zu Verfügung.

Mit `if (!eventObjekt)` wird geprüft, ob der Wert der Variablen bei einer Umwandlung in den Typ Boolean den Wert `false` ergibt. Eine solche Abfrage ist hier möglich, weil `eventObjekt` entweder ein Objekt enthält oder, falls der Handlerfunktion nichts übergeben wird, mit dem Wert `undefined`initialisiert wird. Dieser ergibt in Boolean umgewandelt `false`.

Eine gleichwertige Alternativ-Schreibweise nutzt den `||`-Operator. Intern funktioniert dies wie die besagte `if`-Anweisung : Es

wird geprüft, ob ein Funktionsparameter übergeben wurde. Falls nicht, wird versucht, das Event-Objekt über `window.event` anzusprechen. Das Ziel ist ebenfalls vereinheitlichter Zugriff auf das Event-Objekt über die Variable `eventObjekt`.

```
function handlerfunktion (eventObjekt) {
   eventObjekt = eventObjekt || window.event;
   window.alert("Es ist ein Ereignis vom Typ " + eventObjekt.type + " passiert.");
}
```

Der Oder-Operator `||` überprüft, ob der Wert links `true` ergibt, also der Parameter`eventObjekt` gesetzt wurde. Wenn dies der Fall ist, ergibt der Ausdruck den Wert von `eventObjekt`und es wird quasi `eventObjekt = eventObjekt` ausgeführt. Dabei passiert selbstverständlich nichts, die Variable wird mit sich selbst überschrieben.

Interessant ist der andere Fall, wenn `eventObjekt` im Internet Explorer den Wert `undefined`hat, weil kein Wert für den Parameter übergeben wurde (siehe oben). Dann ist das Ergebnis des Ausdrucks der Wert rechts vom `||`-Operator. Somit wird die Zuweisung `eventObjekt = window.event` ausgeführt. Durch diese Oder-Verzweigung ist das Event-Objekt in jedem Fall in der Variable `eventObjekt` gespeichert.

Welche der Schreibweise Sie verwenden, bleibt ihnen überlassen, denn sie erfüllen dieselbe Funktion. Die erste ist klarer und leicht verständlich, die zweite ist kürzer, erfordert jedoch das Verständnis des `||`-Operators.

In den obigen Beispielen wird das Event-Objekt in der Variable mit dem sprechenden Namen`eventObjekt` gespeichert. Die Namenswahl bleibt selbstverständlich Ihnen überlassen. Es hat sich eingebürgert, diese Variable der Kürze halber `e` zu nennen, um Tipparbeit zu sparen. Wenn in einer Handler-Funktion eine Variable `e` auftaucht, dann ist darin in der Regel das Event-Objekt gespeichert. Sie könnten gleichermaßen schreiben:

```
function handlerfunktion (e) {
   e = e || window.event;
   window.alert("Es ist ein Ereignis vom Typ " + e.type + " passiert.");
}
```

Unterdrücken der Standardaktion des Ereignisses

Viele Ereignisse im Dokument haben eigentümliche Auswirkungen. Ein Beispiel: Wenn der Anwender auf einen Link klickt, so tritt ein `click`-Ereignis ein. Das bringt den Browser dazu, dem Link zu folgen und zum angegebenen Linkziel (der URI) zu navigieren. Das bedeutet, dass der Browser die Ressource vom Webserver herunterlädt und anzeigt. Ein weiteres Beispiel: Das Aktivieren eines Absende-Buttons eines Formulars löst ein `submit`-Ereignis aus, das zur Übertragung des Formulars an den Webserver führt.

Der Browser behandelt also standardmäßig gewisse Ereignisse und führt die sogenannte**Standardaktion** (englisch *default action*) aus, ohne dass der Seitenautor eine entsprechende JavaScript-Logik definiert hat.

Beim Unobtrusive JavaScript versieht man z.B. bestehende Hyperlinks mit einer JavaScript-Logik. Die ursprüngliche Funktionalität des Links will man dann zumeist unterbinden: Beim Klick auf den Link soll nur das Script ausgeführt werden, nicht mehr das Linkziel angesprungen werden.

Angenommen, wir haben folgenden Link:

```
<a href="bilder/bild.jpg" id="vollbildlink">Bild in Originalgröße ansehen</a>
```

Mit JavaScript soll diesem Link nun ein `click`-Handler hinzugefügt werden, der das verlinkte Bild im aktuellen Dokument einblendet, anstatt das Dokument durch das Bild auszuwechseln und das Bild einzeln anzuzeigen. Wie dieses Einblenden umgesetzt wird, interessiert uns an dieser Stelle nicht, sondern nur das Unterdrücken der Standardaktion.

Im traditionellen Event-Handling wird die Standardaktion unterdrückt, indem die Handler-Funktion`false` als Ergebnis zurückgibt. Am Ende der Funktion wird daher die Anweisung `return false;`notiert.

```
function zeigeVollbild () {
   // Blende das Bild ein, auf das der Link zeigt.
   // ... (an dieser Stelle uninteressant) ...

   // Unterdrücke schließlich die Standardaktion:
   return false;
}

// Registriere Event-Handler
document.getElementById("vollbildlink").onclick = zeigeVollbild;
```

Beachten Sie, dass mit der `return`-Anweisung die Funktion beendet wird. Code, der auf diese Anweisung folgt, wird nicht ausgeführt. Es sei denn, die `return`-Anweisung ist z.B. durch eine `if`-Anweisung gekapselt und wird nicht in jedem Fall ausgeführt.

Wenn Sie kein `return false` notieren, führt der Browser automatisch die Standardaktion aus. Sie müssen ihn also nicht mit einem `return true` oder auf andere Art dazu bringen – sie können die Standardaktion lediglich verhindern.

Neben `return false` gibt es modernere Techniken, um die Standardaktion zu verhindern. Der DOM-Standard, auf den wir später noch zu sprechen kommen, bietet eine Methode namens`preventDefault` beim Event-Objekt, mit der sich die Standardaktion unterdrücken lässt. Das obige Beispiel könnte auch folgendermaßen aussehen:

```
function zeigeVollbild (eventObjekt) {
   // Browserübergreifender Zugriff auf das Event-Objekt
   if (!eventObjekt) eventObjekt = window.event;

   // Unterdrücke die Standardaktion durch Aufruf von preventDefault:
   e.preventDefault();
   // Fehler im Internet Explorer < 9!

   // Blende das Bild ein, auf das der Link zeigt.
   // ... (an dieser Stelle uninteressant) ...
};
```

Der Vorteil von `preventDefault` ist, dass es im Gegensatz zu `return false` auch mitten in der Handler-Funktion aufgerufen werden kann, ohne sie gleichzeitig zu beenden. Das Beispiel demonstriert dies.

Der Nachteil ist, dass der Internet Explorer diese standardisierte Methode erst ab Version 9 kennt. Er hat jedoch eine gleichwertige Boolean-Eigenschaft des Event-Objekts namens `returnValue`. Weist man dieser den Wert `false` zu, so wird die Standardaktion unterbunden. Um auch ältere Internet Epxlorer zu unterstützen, kann die Existenz der `preventDefault`-Methode abgefragt werden. Existiert diese nicht, wird alternativ die Eigenschaft `returnValue` gesetzt:

```
function zeigeVollbild (eventObjekt) {
   if (!eventObjekt) eventObjekt = window.event;

   // Existiert die Methode preventDefault? Dann rufe sie auf.
   if (eventObjekt.preventDefault) {
      // W3C-DOM-Standard
      eventObjekt.preventDefault();
   } else {
      // Andernfalls setze returnValue
      // Microsoft-Alternative für Internet Explorer < 9
      eventObjekt.returnValue = false;
   }

   // Blende das Bild ein, auf das der Link zeigt.
   // ... (an dieser Stelle uninteressant) ...
};
```

Diese Vorgehensweise sei hier der Vollständigkeit halber erwähnt. Wenn sie Ihnen unnötig kompliziert erscheint, so können Sie sich mit dem herkömmlichen `return false` zufrieden geben, das die Aufgabe hinreichend erfüllt. Sie müssen allerdings beachten, dass mit `return false` die Handler-Funktion beendet wird.

Der Event-Fluss: Bubbling

Bisher haben wir erfahren, dass Ereignisse bei bestimmten Elementen passieren. Dort können wir sie überwachen, indem wir Handler registrieren. Tritt das Ereignis bei diesem Element ein, wird die Handler-Funktion ausgelöst.

Die Wirklichkeit ist etwas komplizierter. Die Verarbeitung eines Ereignisses verläuft in **drei Phasen**, die nacheinander durchlaufen werden. Davon lernen wir nun eine zweite kennen. Die dritte ist weniger wichtig und braucht Sie zum Einstieg erst einmal nicht interessieren – Sie finden ihre Beschreibung unter Capturing beschrieben.

Ein Ereignis passiert bei einem Element, dem sogenannten **Zielelement** (englisch *target element*), und löst dort alle Handler aus, die für das entsprechende Ereignis registriert wurden – soweit waren wir bereits. Diese bereits bekannte Phase nennt sich entsprechend **Ziel-Phase**.

Mit dieser Phase ist die Ereignis-Verarbeitung nicht zuende, denn anschließend steigt das Ereignis im DOM-Elementenbaum auf. Dieser Vorgang nennt sich **Bubbling**. Das Wort ist abgleitet von *bubble*, Englisch für Blase. Bubbling ist demnach das **Aufsteigen** der Events wie beispielsweise Luftblasen im Wasser.

Dieses Aufsteigen bedeutet, dass die entsprechenden Handler auch beim Eltern-Element des Zielelements ausgeführt werden, dann bei dessen Eltern-Element und so weiter, bis das Ereignis schließlich den obersten `document`-Knoten erreicht hat. Das Ereignis bewegt sich also nach oben im Elementbaum, durchläuft alle Vorfahrenelemente des Zielelements und löst auf diesem Weg alle entsprechenden Handler aus. Dieser Vorgang wird entsprechend **Bubbling-Phase** genannt.

Das mag für den Anfang unverständlich klingen, der Sinn und die Funktionsweise des Bubblings sind aber schnell erfasst. Nehmen wir folgenden HTML-Code:

```
<p id="absatz">
   Dies ist ein Beispiel-Element mit einem
   <strong id="wichtig">
      wichtigen Text
   </strong>.
</p>
```

Nehmen wir ferner an, dass das `p`-Element auf traditionelle Weise einen `click`-Handler bekommt:

```
function klickverarbeitung () {
   window.alert("Der Absatz wurde geklickt!");
}
document.getElementById("absatz").onclick = klickverarbeitung;
```

Das `p`-Element wird vom Browser als rechteckige Box dargestellt. Bei einem Klick irgendwo in diese Box soll die Handler-Funktion ausgeführt werden.

Wenn der Anwender auf das Wort »Beispiel-Element« klickt, ist das `p`-Element das Zielelement des Ereignisses. Wenn man hingegen auf »wichtigen Text« klickt, so ist das `strong`-Element das Zielelement des Ereignisses, nicht das p-Element! Denn dieser Text liegt in erster Linie im `strong`-Element und nur indirekt im `p`-Element. Aus Sicht des DOM-Baumes ist der Text ein Textknoten, der ein Kindknoten des `strong`-Elementknotens ist.

Nichtsdestoweniger erwartet man, dass ein Klick auf die Box des `strong`-Elements ebenfalls den`click`-Handler beim `p`-Element auslöst. Und dies ist auch der Fall – dafür sorgt das Bubbling! Das Ereignis, das ursprünglich beim `strong`-Element passiert ist, steigt nämlich auf, sodass der Handler des`p`-Elements ausgeführt wird.

Das Bubbling ist also meist erwünscht, damit bei einem Element Ereignisse überwacht werden können, selbst wenn sie ursprünglich bei Kindelementen passieren. Wenn Sie aber nicht damit rechnen, dass Ereignisse aufsteigen, so kann das Bubbling zu einiger Verwirrung führen und Sie werden sich wundern, woher plötzlich gewisse Ereignisse stammen.

Nicht alle Ereignisse steigen auf, denn für manche Ereignisse wäre es kontraproduktiv, wenn sie zentrale Handler auslösen würden.

Verarbeitendes Element und Zielelement

Durch das beschriebene Bubbling ist es möglich, dass sich das Element, bei dem ein Ereignis ursprünglich passiert ist, von dem unterscheiden kann, dessen Handler gerade aufgerufen wird. Es ist möglich, dass das Element, das das Ereignis verarbeitet, im DOM-Elementenbaum oberhalb vom Zielelement liegt. Das Ereignis steigt in dem Fall vom Zielelement auf und löst bei einem anderen Element die Handler-Funktion aus.

In vielen Fällen will man in der Handler-Funktion auf beide beteiligten Elemente zugreifen, sofern sie sich unterscheiden.

Beginnen wir mit dem Zugriff auf das **verarbeitende Element**, bei dem die Handler-Funktion registriert wurde: Das Element kann in der Handler-Funktion über das Schlüsselwort `this` angesprochen werden, denn die Handler-Funktion wird im Kontext dieses Elementobjektes ausgeführt.

Das obige Beispiel wird wieder aufgegriffen:

```
<p id="absatz">
   Dies ist ein Beispiel-Element mit einem
   <strong id="wichtig">
         wichtigen Text
   </strong>.
</p>
```

Dem Absatz wird wieder ein `click`-Handler zugewiesen:

```
function klickverarbeitung () {
   window.alert("Element vom Typ " + this.nodeName + " wurde geklickt!");
}
document.getElementById("absatz").onclick = klickverarbeitung;
```

Innerhalb der Handler-Funktion können wir über `this` auf das `p`-Element zugreifen. Im Beispiel wird auf dessen Eigenschaft `nodeName` ausgegeben, welche den Elementnamen `P` enthält.

Der DOM-Standard sieht eine andere Zugriffsweise vor: Die Eigenschaft `currentTarget` beim Event-Objekt enthält das Element, dessen Handler gerade ausgeführt wird. Der Internet Explorer kennt diese Eigenschaft erst an Version 9. Das besagte `this` ist in älteren Internet-Explorer-Versionen die einzige Möglichkeit, auf das fragliche Element zuzugreifen. Der Einfachheit halber können Sie browserübergreifend `this` verwenden.

Der eindeutige Zugriff auf das **Zielelement** gestaltet sich etwas schwieriger. Der DOM-Standard definiert die Eigenschaft `target` beim Event-Objekt. Diese kennen alle modernen Browser, doch der Internet Explorer unterstützt diese Eigenschaft erst ab Version 9. Ältere Versionen kennen eine äquivalente Eigenschaft namens `srcElement`. Mithilfe einer Fähigkeitenweiche nehmen wir eine Vereinheitlichung vor, sodass das Zielelement in allen Browsern über eine Variable ansprechbar ist – wir kennen dieses Vorgehensweise bereits vom Zugriff auf das Event-Objekt.

```
function klickverarbeitung (eventObjekt) {
   if (!eventObjekt) eventObjekt = window.event;

   // Zugriff auf das Zielelement
   if (eventObjekt.target) {
      // W3C-DOM-Standard
      var target = eventObjekt.target;
   } else {
      // Microsoft-Alternative für Internet Explorer < 9
      var target = eventObject.srcElement;
   }
```

```
   window.alert(
      "Das Ereignis passierte ursprünglich beim Element " + target.nodeName +
      " und wird vom Element " + this.nodeName + " verarbeitet.");
   );
}
```

Falls die Eigenschaft `target` des Event-Objektes gefüllt ist, legen wir in der lokalen Variable`target` eine Referenz darauf an. Andernfalls, das betrifft den Internet Explorer, wird die Eigenschaft`srcElement` verwendet.

Wie beim Zugriff auf das Event-Objekt erlaubt der `||`-Operator eine Kurzschreibweise. Das Event-Objekt wird zudem unter dem Kurznamen `e` gespeichert. So kommen wir zu einem Schema, dem viele Handler-Funktionen entsprechen:

```
function klickverarbeitung (e) {
   // Vereinheitlichung von Event-Objekt und Zielelement
   e = e || window.event;
   var target = e.target || e.srcElement;

   // Nutzlast
   window.alert(
      "Das Ereignis passierte ursprünglich beim Element " + target.nodeName +
      " und wird vom Element " + this.nodeName + " verarbeitet.");
   );
}
```

Interaktives Beispiel

Das folgende Beispiel ist ein `div`-Element, an dem ein `mouseover`-Handler registriert wurde. In dem `div`-Element sind verschiedene Elemente verschachtelt. Bewegen Sie die Maus über deren Boxen, so werden `mouseover`-Ereignisse an diesen ausgelöst. Sie steigen im DOM-Baum auf und werden beim gemeinsamen `div`-Elemente verarbeitet. Unter dem `div` werden das Zielelement und das verarbeitende Element des Ereignisses ausgegeben. Das verarbeiten Element ist immer dasselbe, nämlich das `div`-Containerelement.

Ein einfacher Textabsatz ohne weiteres.

1. Listenelement mit **wichtigem Text**.
2. Listenelement mit *hervorgehobenem* Text.

Das `mouseover`-Ereignis passierte ursprünglich beim Element **P** und wird vom Element **DIV**verarbeitet.

Der JavaScript-Quellcode dieses Beispiels sieht so aus:

```
var targetArea = document.getElementById('target-area');
var output = document.getElementById('target-output');
targetArea.onmouseover = function (e) {
   e = e || window.event;
   var target = e.target || e.srcElement;
   output.innerHTML = "Das <code>mouseover</code>-Ereignis passierte ursprünglich beim Element " +
      "<strong><code>" + target.nodeName + "</code></strong> und wird vom Element " +
      "<strong><code>" + this.nodeName + "</code></strong> verarbeitet.";
};
```

Der Code geht davon aus, dass zwei Elemente mit den IDs `target-area` und `target-output`definiert sind und das Script später im Dokument notiert ist, sodass es Zugriff auf die Elemente hat.

Kontrolle über den Event-Fluss: Bubbling verhindern

Es gibt Fälle, in denen das Bubbling nicht gewünscht ist. Beispielsweise wenn zwei verschachtelte Elemente dasselbe Ereignis überwachen, aber nur der Handler des inneren Elements aktiv werden soll, wenn dieses das Ziel des Ereignisses ist. In einer Handler-Funktion können Sie deshalb das weitere Aufsteigen des Ereignisses im Elementenbaum verhindern.

Folgendes bekannte Beispiel mit verschachtelten Elementen soll dies illustrieren:

```
<p id="absatz">
   Dies ist ein Beispiel-Element mit einem
   <strong id="wichtig">
      wichtigen Text
   </strong>.
</p>
```

Das `strong`-Element steckt hier im `p`-Element. Bei beiden Elementen wird ein `click`-Handler registriert:

```
function absatzKlick () {
   window.alert("Klick auf das p-Element");
}
document.getElementById("absatz").onclick = absatzKlick;

function wichtigKlick () {
   window.alert("Klick auf das strong-Element");
}
document.getElementById("wichtig").onclick = wichtigKlick;
```

Bei einem Klick auf die Fläche des `strong`-Elements (»wichtigen Text«) werden beide Handler-Funktionen ausgeführt, denn das Ereignis steigt vom `strong`-Element zum `p`-Element auf.

Dieses Aufsteigen können Sie in der Handler-Funktion des `strong`-Elementes (`wichtigKlick`) verhindern. Der DOM-Standard definiert dafür die Methode `stopPropagation` (englisch: stoppe die Verbreitung bzw. Weitergabe des Ereignisses) beim Event-Objekt. Ein Aufruf dieser Methode unterbricht den Event-Fluss und verhindert damit das (weitere) Aufsteigen.

Der Internet Explorer kennt diese Methode erst ab Version 9. Ältere Versionen verfügen über eine gleichwertige Boolean-Eigenschaft beim Event-Objekt. Diese trägt den Namen `cancelBubble`(englisch: breche das Aufsteigen ab). Weisen Sie dieser Eigenschaft den Wert `true` zu, um das Aufsteigen des Ereignisses abzubrechen.

Wieder einmal nutzen wir eine Fähigkeitenerkennung, die die Verfügbarkeit der standardisierten Methode `stopPropagation` prüft und im Fehlerfalle auf die Microsoft-Alternative `cancelBubble`zurückfällt.

Die Handler-Funktion `wichtigKlick` wird wie folgt modifiziert:

```
function wichtigKlick (eventObekt) {
   if (!eventObjekt) eventObjekt = window.event;

   // Stoppe das Aufsteigen des Ereignisses
   if (eventObjekt.stopPropagation) {
      // W3C-DOM-Standard
      eventObjekt.stopPropagation();
   } else {
      // Microsoft-Alternative für Internet Explorer < 9
      eventObjekt.cancelBubble = true;
  }

   window.alert("Klick auf das strong-Element. Das Aufsteigen des Ereignisses wird unterbunden!");
}
```

Damit können verschachtelte Elemente denselben Ereignistyp überwachen, im Beispiel `click`. Obwohl das eine Element in dem anderen enthalten ist und üblicherweise in dessen Grenzen dargestellt wird, übernimmt es die Ereignis-Verarbeitung selbstständig. Der Handler des äußeren Elements, im Beispiel `absatzKlick` beim `p`-Element, wird nur bei Klicks ausgeführt, die auf seine Fläche zielen, ausgenommen die Fläche des inneren Elements.

JavaScript: Fortgeschrittene Ereignisverarbeitung

Nachteile des traditionellen Event-Handlings und Alternativen

Das traditionelle Event-Handling basiert darauf, dass ein Funktionsobjekt in einer Eigenschaft des Elementobjektes gespeichert wird. Wir erinnern uns an das Schema `element.onevent = handlerfunktion`.

Der Vorteil dieses Schema ist seine Einfachheit und Verständlichkeit. Will man ein Ereignis überwachen, schreibt man bloß die Handler-Funktion in eine entsprechende Element-Eigenschaft. Der größte Nachteile ist jedoch folgender: Es kann nur *eine* Handler-Funktion zugleich registriert werden. Denn in der Eigenschaft kann nur eine Funktion gespeichert werden, und weist man eine andere Funktion zu, überschreibt man die erste.

In manchen Fällen mag es ausreichen, dass man je Element für einen Ereignistyp nur eine Handlerfunktion definieren kann. Diese kann schließlich weitere Funktionen aufrufen, sodass nicht der gesamte auszuführende Code direkt in dieser einen Funktion stehen muss. Doch insbesondere wenn verschiedene Scripte zusammenarbeiten, besteht die Gefahr, dass sie beim traditionellen Event-Handling einander in die Quere kommen.

Es ist es durchaus möglich, mehrere Handler-Funktionen zu notieren, ohne letztlich vom traditionellen Schema abzuweichen. Dazu sind allerdings Helferscripte nötig, deren Funktionsweise nur für JavaScript-Kenner zu verstehen ist. Bevor wir auf solche »Workarounds« eingehen, wenden wir uns den fortgeschrittenen Modellen für Event-Handling zu.

Event-Handling gemäß dem W3C-Standard DOM Events

Das bisher beschriebene traditionelle Schema stammt aus den Anfangstagen von JavaScript. Der Browserhersteller und JavaScript-Erfinder Netscape erfand das Schema einst und andere Browser übernahmen es im Zuge ihrer JavaScript-Unterstützung.

Die Entwicklung ging jedoch weiter: Bei der Standardisierung des Event-Handlings verwarf das WWW-Konsortium das traditionelle Event-Handling. Der entsprechende DOM-Standard sieht ein anderes Modell vor: Alle Elementobjekte und weitere zentrale Objekte besitzen die Methode`addEventListener` (englisch: Ereignis-Überwacher hinzufügen). Will man dem Element einen Event-Handler zuweisen, so ruft man diese Methode auf.

Event-Handler registrieren: `addEventListener`

Das standardisierte Schema enthält ebenfalls die drei Bestandteile Elementobjekt, Ereignistyp und Handler-Funktion. Es lautet folgendermaßen:

```
element.addEventListener("event", handlerfunktion, capturing);
```

Die Methode erwartet also drei Parameter:

1. Der erste Parameter ist ein String und enthält den **Ereignistyp**. Beispiele für den ersten Parameter sind `"click"`, `"mouseover"`, `"load"`, `"submit"` und so weiter.
2. Der zweite Parameter ist der Name der **Handler-Funktion**, genauer gesagt ein Ausdruck, der ein Funktionsobjekt ergibt.
3. Der dritte Parameter bestimmt, für welche **Event-Phase** der Handler registriert werden soll. Es handelt sich um einen Boolean-Parameter, d.h. sie können `true` oder `false` notieren. `false`steht für die bereits bekannte Bubbling-Phase, `true` für die noch nicht behandelte und weniger wichtige *Capturing-Phase* (siehe Capturing). Die genaue Bedeutung des dritten Parameters wird erst später erklärt werden. Standardmäßig sollten Sie hier `false` notieren.

Das folgende Beispiel kennen wir bereits vom traditionellen Event-Handling. Beim fertigen Laden des HTML-Dokuments wird automatisch ein `click`-Handler bei einem Textabsatz registriert. Dieses Mal nutzen wir die standardisierte Methode `addEventListener`, die denselben Zweck erfüllt:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Beispiel für Event-Handling gemäß dem W3C DOM</title>
<script type="text/javascript">

window.addEventListener("load", start, false);

function start () {
   var pElement = document.getElementById("interaktiv");
   pElement.addEventListener("click", klickverarbeitung, false);
}

function klickverarbeitung () {
   document.getElementById("interaktiv").innerHTML +=
      " Huhu, das ist von Javascript eingefügter Text.";
}

</script>
</head>
<body>

<p id="interaktiv">
   Dies ist ein einfacher Textabsatz, aber mithilfe von JavaScript können wir ihn
   interaktiv gestalten. Klicken Sie diesen Absatz doch einfach mal mit der Maus an!
</p>

</body>
</html>
```

Folgende Anweisungen haben sich geändert:

Aus der Zuweisung `window.onload = start;` ist der Aufruf`window.addEventListener("load", start, false);` geworden. Wenngleich `window` kein Elementobjekt ist, bietet es dennoch die Methode `addEventListener` an.

Aus der Zuweisung `document.getElementById("interaktiv").onclick = klickverarbeitung;` sind zwei

geworden. In der ersten Anweisung speichern wir das Elementobjekt des Absatzes in einer Variable zwischen:

```
var pElement = document.getElementById("interaktiv");
```

In der zweiten Anweisung wird schließlich die Handler-Funktion registriert:

```
pElement.addEventListener("click", klickverarbeitung, false);
```

Sie können die Methoden-Aufrufe natürlich auch verketten, anstatt eine Hilfsvariable zu verwenden. Das sähe schematisch so aus:`document.getElementById(…).addEventListener(…)`. Schließlich gibt `getElementById` im Regelfall ein Elementobjekt zurück, dessen Methoden Sie direkt ansprechen können. Aus Gründen der Lesbarkeit und Verständlichkeit wurde diese Kette im Beispiel in zwei Anweisungen gesplittet.

Das obige Beispiel funktioniert in allen modernen Browsern – jedoch nicht im Internet Explorer vor der Version 9. Die älteren Internet-Explorer-Versionen unterstützen den W3C-DOM-Standard noch nicht. Die Methode `addEventListener` ist diesen Browsern schlicht unbekannt.

Der Internet Explorer unterstützt den Events-Standard erst ab Version 9. Ältere Versionen unterstützen stattdessen ein eigenes, proprietäres Modell, das im folgenden Abschnitt vorgestellt wird.

Der Hauptvorteil von `addEventListener` ist, dass Sie für ein Element **mehrere Handler-Funktionen** für denselben Ereignistyp registrieren können. Beim obigen Beispiel können wir die `start`-Funktion so anpassen, dass beim `p`-Element zwei Handler statt bloß einer registriert werden:

```
function start () {
   var pElement = document.getElementById("interaktiv");
   pElement.addEventListener("click", meldung1, false);
   pElement.addEventListener("click", meldung2, false);
}

function meldung1 () {
   window.alert("Erste Handler-Funktion ausgeführt!");
}

function meldung2 () {
   window.alert("Zweite Handler-Funktion ausgeführt!");
}
```

Es werden zwei Handler-Funktionen namens `meldung1` und `meldung2` definiert. Mithilfe von`addEventListener` werden sie beide als `click`-Handler registriert. Wenn Sie auf den Textabsatz klicken, dann sollten nacheinander zwei JavaScript-Meldefenster erscheinen – und zwar in der Reihenfolge, in der die Handler mittels `addEventListener` registriert wurden.

Event-Handler entfernen: removeEventListener

Um die mit `addEventListener` registrierten Handler wieder zu **entfernen**, gibt es die Schwestermethode `removeEventListener` (englisch: Ereignis-Empfänger entfernen). Die Methode erwartet dieselben Parameter, die `addEventListener` beim Registrieren bekommen hat: Einen String mit dem Ereignistyp, die zu löschende Handler-Funktion und schließlich einen Boolean-Wert für die Event-Phase.

Um beide im Beispiel definierten Handler für das `p`-Element (nämlich `meldung1` und `meldung2`) wieder zu entfernen, notieren wir:

```
function beenden () {
   pElement.removeEventListener("click", meldung1, false);
   pElement.removeEventListener("click", meldung2, false);
}
```

Event-Handling gemäß Microsoft für ältere Internet Explorer

Microsoft hat schon früh für seinen Internet Explorer eine Alternative zum unzureichenden traditionellen Event-Handling eingeführt, welches seinerseits vom damaligen Konkurrenten Netscape erfunden wurde. Das W3C-DOM wurde erst später standardisiert und wurde erst Jahre später im Internet Explorer 9 eingebaut. Das Microsoft-Modell wiederum wird nur vom Internet Explorer verstanden. Es handelt sich hier also um eine Sonderlösung, die nur noch für ältere Internet-Explorer-Versionen interessant ist, welche den DOM-Standard nicht umsetzen.

Microsofts Modell teilt einige Fähigkeiten mit `addEventListener` und`removeEventListener`, funktioniert im Detail jedoch anders und bringt einige Eigenheiten und Schwierigkeiten mit sich.

Event-Handler registrieren: attachEvent

Im Microsoft-Modell besitzt jedes Elementobjekt sowie einige zentrale Objekte die Methode`attachEvent` zum Registrieren von Event-Handlern. Das Schema lautet folgendermaßen:

```
element.attachEvent("onevent", handlerfunktion);
```

Die Methode erwartet zwei Parameter:

1. Der erste Parameter ist ein String und enthält den **Ereignistyp** mit der Vorsilbe `on`. Beispiele für den ersten Parameter sind `"onclick"`, `"onmouseover"`, `"onload"`, `"onsubmit"` und so weiter.
2. Der zweite Parameter ist der Name der **Handler-Funktion** (ein Ausdruck, der ein Funktionsobjekt ergibt).

Wir greifen das bekannte Beispiel auf, das wir bereits nach dem traditionellem Modell und nach dem W3C-Modell umgesetzt haben, und setzen es mit dem Microsoft-Modell um:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>Beispiel für Event-Handling gemäß dem W3C DOM</title>
<script type="text/javascript">

window.attachEvent("onload", start);

function start () {
   var pElement = document.getElementById("interaktiv");
   pElement.attachEvent("onclick", klickverarbeitung);
}

function klickverarbeitung () {
   document.getElementById("interaktiv").innerHTML +=
      " Huhu, das ist von Javascript eingefügter Text.";
}

</script>
</head>
<body>

<p id="interaktiv">
   Dies ist ein einfacher Textabsatz, aber mithilfe von JavaScript können wir ihn
   interaktiv gestalten. Klicken Sie diesen Absatz doch einfach mal mit der Maus an!
</p>

</body>
</html>
```

Das Beispiel hat sich gegenüber dem W3C-Modell nur geringfügig geändert. Anstelle von`window.addEventListener(…)` wurde `window.attachEvent(…)` notiert, dasselbe bei`pElement.attachEvent(…)`.

Der Ereignistyp im ersten Parameter enthält nun den Präfix `on` vorangestellt: Aus `"load"` wird`"onload"`, aus `"click"` wird `"onclick"`. Der dritte Parameter, der die Event-Phase spezifiert, fällt weg – denn Microsofts Modell unterstützt nur das Registrieren in der Bubbling-Phase.

Auch mit `attachEvent` können Sie verschiedene Handler für denselben Ereignistyp definieren. Das obige Beispiel wird entsprechend angepasst:

```
function start () {
   var pElement = document.getElementById("interaktiv");
   pElement.attachEvent("onclick", meldung1);
   pElement.attachEvent("onclick", meldung2);
}

function meldung1 () {
   window.alert("Erste Handler-Funktion ausgeführt!");
}

function meldung2 () {
   window.alert("Zweite Handler-Funktion ausgeführt!");
}
```

Das Beispiel enthält nichts neues, die Aufrufe von `addEventListener` wurden auf die besagte Weise durch `attachEvent` ausgetauscht.

Event-Handler entfernen: detachEvent

Auch das Microsoft-Modell bietet eine Methode, um registrierte Handler wieder zu entfernen. Sie nennt sich `detachEvent` und erwartet dieselben Parameter wie sein Gegenstück `attachEvent`.

Um die besagten `click`-Handler `meldung1` und `meldung2` wieder zu entfernen, notieren wir:

```
function beenden () {
   pElement.detachEvent("onclick", meldung1);
   pElement.detachEvent("onclick", meldung2);
}
```

Eigenheiten des Microsoft-Modell

Das Microsoft-Modell bringt eine erfreuliche und eine unerfreuliche Besonderheit mit sich:

- Bei der Verwendung von `attachEvent` gestaltet sich der Zugriff auf das Event-Objekt einfacher, als wir es vom traditionellen Event-Handling gewöhnt sind. Dort war der Zugriff über`window.event` nötig. Bei der Benutzung von `attachEvent` wird das Event-Objekt der Handler-Funktion als Parameter übergeben, wie wir es aus anderen Browsern gewohnt sind und wie es auch beim im DOM-Standard vorgeschrieben ist.
- Der Zugriff auf das verarbeitende Element ist nicht möglich. Beim traditionellen Event-Handling hatten wir `this` kennengelernt, um das Element anzusprechen, bei dem die gerade ausgeführte Handler-Funktion registriert wurde. Das ist im Zusammenhang mit `attachEvent` nicht möglich, denn `this` zeigt nicht auf das gewünschte Objekt, sondern stets auf das globale Objekt `window`– und ist damit unbrauchbar.
- Im Gegensatz zum DOM-Standard kennt das Microsoft-Modell keine Capturing-Phase.

Browserübergreifendes Event-Handling

Ausgangslage

Wir haben drei Modelle und deren Detailunterschiede kennengelernt. Das mag Sie verwirrt haben und Sie werden sich sicher fragen, welches Sie nun in der Praxis ohne Bedenken anwenden können. Das Fazit lautet leider: Keines.

- Das traditionelle Event-Handling reicht in einfachen Fällen aus, Sie werden aber sehr schnell an dessen Grenzen stoßen.
- DOM Events ist zweifelsohne das leistungsfähigste und vielseitigste, der gravierende Nachteil ist jedoch, dass Internet-Explorer-Versionen den W3C-Standard nicht umsetzen. Die Versionen vor 9 sind derzeit noch weit verbreitet.
- Das Microsoft-Modell taugt zwar als Ausweichlösung für ältere Internet Explorer. Es bringt jedoch Einschränkungen mit sich, die einer Korrektur bedürfen. Und manche unüberbrückbaren Unterschiede führen dazu, dass es nicht gleichwertig zum W3C-Modell eingesetzt werden kann.

`addEvent`-Helferfunktionen

Um relativ komfortabel browserübergreifend Ereignisse verarbeiten zu können, benötigen wir eine Helferfunktion, die eine Vereinheitlichung vornimmt und gewisse Browserunterschiede nivelliert. Eine solche alleinstehende Funktion zum Registrieren von Event-Handlern wird in der Regel `addEvent`genannt

Die Entwicklung einer solchen `addEvent`-Funktion ist eine Wissenschaft für sich. ModerneJavaScript-Bibliotheken nutzen äußerst ausgefeilte Umsetzungen. Diese basieren auf jahrelanger Forschung, um viele Sonderfälle abzudecken und dem Anwender das Event-Handling durch Vereinheitlichung aller relevanter Browserunterschiede zu vereinfachen.

Diese Bibliotheken bringen jedoch eine eigene Arbeitsweise sowie unzählige weitere Funktionen mit sich. Daher seien hier zwei isolierte Helferscripte vorgestellt.

Einfaches, oftmals ausreichendes `addEvent`

Ein einfaches Beispiel ist A Good Enough addEvent von Austin Matzko. Ziel ist es, mehrere Event-Handler für einen Typ bei einem Element registrieren zu können. Die Funktion verwendet`addEventListener` (DOM-Standard) oder `attachEvent` (Microsoft-Modell) je nach Verfügbarkeit.

```
function addEvent (obj, type, fn) {
   if (obj.addEventListener) {
      obj.addEventListener(type, fn, false);
   } else if (obj.attachEvent) {
      obj.attachEvent('on' + type, function () {
         return fn.call(obj, window.event);
      });
   }
}
```

Ein Anwendungsbeispiel:

```
<p id="beispielabsatz">Klick mich!</p>
<script type="text/javascript">
function absatzKlick () {
        alert("Der Absatz wurde geklickt!");
}
addEvent(document.getElementById("beispielabsatz"), "click", absatzKlick);
</script>
```

Da die beiden Modelle im Detail zueinander inkompatibel sind, sind im `attachEvent`-Zweig einige Anpassungen vonnöten. Diese müssen Sie im Detail nicht verstehen, sie seien hier dennoch erklärt:

1. Die übergebene Handler-Funktion wird in einer weiteren gekapselt und eingeschlossen, die an deren Stelle als Handlerfunktion verwendet wird. Der Kontext der Handler-Funktion wird korrigiert. Dadurch ist in der Handler-Funktion mittels `this` der Zugriff auf das verarbeitende Element möglich. (Siehe verarbeitendes Element.)

2. Der Handler-Funktion wird das Event-Objekt als Parameter übergeben. Dadurch fällt in der Handler-Funktion die Vereinheitlichung bem Zugriff darauf weg. (Siehe Zugriff auf das Event-Objekt.)

Kurz gesagt sorgen diese Anpassungen dafür, dass Sie browserübergreifend mglichst gleich arbeiten können und die Browserunterschiede ausgeglichen werden.

Diese `addEvent`-Funktion ist wie gesagt sehr einfach gehalten, was unter anderem folgende Nachteile mit sich bringt:

- Es gibt keine äquivalente Funktion zum Entfernen eines Event-Handlers (`removeEvent`).
- Wenn Sie dieselbe Handler-Funktion für einen Ereignistyp beim selben Element mehrfach registrieren, verhalten sich ältere Internet Explorer anders als moderne Browser, die den DOM-Standard unterstützen: Die Handler-Funktion wird nur einmal ausgeführt anstatt so oft, wie Sie den Handler registriert haben. Diese Fall ist allerdings vermeidbar, in der Regel besteht keine Notwendigkeit, einen Handler mehrfach zu registrieren.
- Es werden nur die allerwichtigsten Browserunterschiede vereinheitlicht, auf viele weitere müssen Sie in der Handler-Funktion selbst reagieren.

Flexibles und leistungsfähiges `addEvent`/`removeEvent`

Eine robustere, aber umso kompliziertere Umsetzung der Funktionen `addEvent` und`removeEvent` stammt von Dean Edwards und wurde von Tino Zijdel weiterentwickelt: addEvent() new style.

Dieses Helferscript verwendet die standardisierte Methoden `addEventListener` und`removeEventListener` in den Browsern, in denen sie zur Verfügung stehen. Als Alternative für ältere Internet Explorer wird allerdings nicht das Microsoft-eigene Modell verwendet – zu groß sind die Unterschiede zwischen den beiden Modellen. Stattdessen wird das Registrieren von mehreren Event-Handlern selbst übernommen: Es wird bei jedem Element eine eigene Liste mit Handler-Funktionen für ein Ereignistyp geführt. Beim Eintreten des Ereignisses wird eine Helferfunktion aufgerufen, die diese interne Liste abarbeitet und jede dort verzeichnete Handler-Funktion aufruft.

Neben `addEvent` umfasst das Script auch eine `removeEvent`-Methode. Im Vergleich zum einfachen `addEvent` werden im Internet Explorer zwei weitere Vereinheitlichungen vorgenommen, sodass folgende Techniken browserübergreifend nutzbar sind:

1. In der Handler-Funktion ist mittels `this` der Zugriff auf das verarbeitende Element möglich. (Sieheverarbeitendes Element.)
2. Der Handler-Funktion wird das Event-Objekt als Parameter übergeben. (Siehe Zugriff auf das Event-Objekt.)
3. Zum Unterdrücken der Standardaktion kann browserübergreifend die standardisierte Methode`preventDefault` des Event-Objektes verwendet werden. Eine entsprechende Fähigkeiten-Weiche in der Handler-Funktion ist nicht nötig. (Siehe Unterdrücken der Standardaktion.)
4. Um das Aufsteigen des Ereignisses zu verhindern, kann die standardisierte Methode`stopPropagation` des Event-Objektes browserübergreifend genutzt werden. (Siehe Bubbling verhindern.)

Falls Sie keine umfangreichere JavaScript-Bibliothek verwenden, welche ausgereiftes Event-Handling ermöglicht, so sind Sie mit diesen beiden Helferfunktionen `addEvent` und `removeEvent` gut bedient – Sie sollten sie in ihren Werkzeugkasten aufnehmen.

Browserübergreifendes Event-Handling mit Frameworks

Die besagte Browser-Vereinheitlichung berührt nur die Spitze des Eisberges: Besonders bei der Verarbeitung von Tastatur- und Maus-Ereignissen erwarten den JavaScript-Programmierer noch viel größere Browserunterschiede. Um beispielsweise browserübergreifend die Mausposition relativ zum Dokument auszulesen, ist eine komplexe Fähigkeiten-Weiche nötig. Dasselbe gilt für das Auslesen der gedrückten Taste(n). Für den JavaScript-Einsteiger ist es schwierig und unkomfortabel, für all diese Detailunterschiede Korrekturen einzubauen. Das eigentliche Verarbeiten der Ereignisse rückt durch dieses Browser-Durcheinander (im Englischen »quirks« genannt), in den Hintergrund.

Wenn Sie bei solchem Event-Handling schnell zu Resultaten kommen wollen, sei Ihnen die Verwendung einer JavaScript-Bibliothek empfohlen. Die gängigen Frameworks nehmen Ihnen die Vereinheitlichung beim Event-Handling größtenteils ab.

JavaScript: Onload-Techniken

Nachteile des `load`-Ereignisses

Wir haben bereits die drei Phasen von ereignisbasierten Scripten kennengelernt. Die zweite Phase tritt ein, wenn das Dokument fertig geladen ist und der DOM-Elementenbaum des Dokuments vollständig aufgebaut ist. Sinnvolle Scripte schalten sich automatisch hinzu, ohne dass JavaScript-Code direkt ins HTML eingebettet ist (Unobtrusive JavaScript). Dazu ist es von zentraler Wichtigkeit, das Script zu initialisieren, sobald der Zugriff auf den gesamten DOM-Baum möglich ist, vor allem um Event-Handler bei gewissen Elementen registriert.

Mit dem dokumentweiten `load`-Ereignis (`window.onload`) haben wir eine Funktion nach dem vollständigen Laden des Dokuments ausgeführt. Das `load`-Ereignis tritt jedoch erst dann ein, wenn das gesamte Dokument *mitsamt aller externen Ressourcen* vom Webserver heruntergeladen wurde. Dazu gehören eingebettete Grafiken, Multimedia-Plugins und gegebenenfalls Iframes mit weiteren HTML-Dokumenten. Je nachdem, welche externen Ressourcen eingebunden werden, kann das Dokument zum Zeitpunkt des `load`-Ereignisses schon längst im Browser aufgebaut sein und der Anwender kann es schon größtenteils lesen und bedienen.

Für das Event-Handling interessiert uns nicht, wann alle eingebetteten Ressourcen fertig geladen sind, sondern lediglich, wann der JavaScript-Zugriff auf den DOM-Baum möglich ist. Dieser Zeitpunkt, genannt *DOM ready*, tritt meistens weit vor dem `load`-Ereignis ein. Ein Script sollte dem Dokument so zeitnah wie möglich Interaktivität hinzufügen und notwendige Änderungen vornehmen, damit das Dokument einschließlich der JavaScript-Funktionalität bereits während des Ladens für den Anwender bedienbar ist.

DOMContentLoaded

Der `load`-Event ist aus heutiger Sicht für Unobtrusive JavaScript nicht geeignet. Glücklicherweise gibt es ein Ereignis, das eintritt, sobald der Parser den gesamten HTML-Code eingelesen hat und der komplette DOM-Baum für JavaScripte zugänglich ist: `DOMContentLoaded`.

Die Überwachung dieses Ereignises ist allerdings nur im Zusammenhang mit addEventListener, also dem W3C-Modell möglich. Alle großen Browser, die den DOM-Standard umsetzen, kennen mittlerweile auch `DOMContentLoaded`. Eine Ausnahme bilden der Internet Explorer vor Version 9: Diese älteren Versionen unterstützen weder den DOM-Events-Standard, noch kennen sie das Ereignis`DOMContentLoaded`.

Das folgende Beispiel demonstriert die Überwachung der beiden Ereignisse `load` und`DOMContentLoaded`:

```
function dokumentGeladen (e) {
   alert("Das Ereignis " + e.type + " ist passiert.")
}
document.addEventListener("load", dokumentGeladen, false);
document.addEventListener("DOMContentLoaded", dokumentGeladen, false);
```

Internet Explorer vor Version 9 können dieses Beispiel nicht ausführen, sie kennen die Methode`addEventListener` nicht.

Wenn Sie das Beispiel in ein HTML-Dokument einfügen, so demonstriert es, dass der`DOMContentLoaded`-Event vor dem `load`-Event passiert. Wenn große und zahlreiche externen Ressourcen in das Dokument eingebunden sind, dann kann zwischen beiden Ereignissen viel Zeit vergehen – wertvolle Zeit, die ein Script nicht untätig verstreichen lassen sollte.

Browserübergreifendes `DOMContentLoaded`

Leider gibt es ältere Browser, die das äußerst nützliche `DOMContentLoaded`-Ereignis nicht unterstützen. Ältere Internet

Explorer kennen auch kein gleichwertiges Ereignis. Daher nutzt die verbreitete Sonderlösung verschiedene Microsoft-eigene Techniken, um Rückschlüsse darauf zu ziehen, ob der Elementenbaum bereits vollständig eingelesen wurde. Die genaue Funktionsweise sei hier nicht näher erklärt. Stattdessen sei auf eine browserübergreifende Fertiglösung hingewiesen, die sowohl`DOMContentLoaded` unterstützt als auch einen erprobten Workaround für ältere Internet Explorer enthält: ContentLoaded.js von Diego Perini. Das Script ist abwärtskompatibel, das heißt, wenn alle Stricke reißen, wird auf das robuste `load`-Ereignis zurückgefallen.

Wenn Sie dieses Script einbinden, können Sie eine Funktion folgendermaßen ausführen, sobald der DOM-Baum zur Verfügung steht:

```
function init () {
// Diese Funktion kann auf den gesamten DOM-Baum zugreifen.
}
// Nutzung von Diego Perinis Helferfunktion
ContentLoaded(window, init);
```

Ein solche DOM-Ready-Technik ist in den verbreiteten JavaScript-Frameworks bereits eingebaut. Diese bringen eigene Event-Handling-Systeme mit sich, die das Ausführen von Funktionen erlauben, sobald die DOM-Schnittstelle vollständig nutzbar ist.

Fallstricke bei `DOMContentLoaded`

Das `DOMContentLoaded`-Ereignis wurde ursprünglich vom Browserhersteller Mozilla erfunden. Andere Browserhersteller haben es übernommen, weil sie sie nützlich fanden. Mittlerweile wird die genaue Funktionsweise in HTML5 standardisiert.

Trotz dieser Standardisierung verhalten sich die Browser im Detail noch unterschiedlich. Eine Unstimmigkeit ist die Frage, ob das Ereignis durch den Ladevorgang externer Stylesheets verzögert werden soll.

Beide Varianten ergeben Sinn: Es reicht zwar aus, dass der DOM-Baum komplett eingelesen ist, wenn man Event-Handler registrieren will. Allerdings gibt es auch Scripte, die in der Initialisierungsphase bereits auf die Darstellung gewisser Elemente reagieren sollen – und diese wird üblicherweise durch externe Stylesheets geregelt. Wenn das Script beispielsweise die Größe eines Elements in Erfahrung bringen will, muss der Browser bereits die Stylesheets verarbeitet und die Darstellung berechnet haben.

JavaScript: Effiziente Ereignisverarbeitung

Event-Delegation

Wenn zahlreiche Elemente im Dokument überwacht werden sollen, ist es sehr aufwändig umzusetzen und langsam in der Ausführung, diese herauszusuchen, zu durchlaufen und bei jedem denselben Event-Handler zu registrieren. Bei solchen Aufgabenstellungen können Sie vom Bubbling-Effekt profitieren, das ist das Aufsteigen der Ereignisse im DOM-Baum. Man macht sich die Verschachtelung der Elemente im DOM-Baum zunutze und überwacht die Ereignisse von verschiedenen Elementen bei einem gemeinsamen, höherliegenden Element, zu dem die Ereignisse aufsteigen. Diese Technik nennt sich **Event-Delegation** (englisch *delegation* für Übertragung von Aufgaben). Dabei wird einem zentralen Element die Aufgabe übertragen, die Ereignisse zu verarbeiten, die bei seinen Nachfahrenelementen passieren.

Event-Delegation eignet sich insbesondere dann, wenn viele gleichförmige Elemente in Menüs, Link-Listen, Formularen oder Tabellen JavaScript-Interaktivität benötigen. Ohne Event-Delegation müsste man jedes Element einzeln ansprechen, um dort immer denselben Event-Handler zu registrieren.

Nehmen wir beispielsweise eine Liste mit Links zu Bildern. Wenn JavaScript aktiv ist, soll das Vollbild dokumentintern eingeblendet werden. Ein ähnliches Beispiel hatten wir bereits beim Unterdrücken der Standardaktion – die Umsetzung der Einblendung bleibt weiterhin ausgeklammert.

Wir gehen von folgendem HTML-Gerüst aus:

```
<ul id="bilderliste">
<li><a href="bilder/bild1.jpg"><img src="bilder/thumbnail1.jpg" alt="">
   Ebru und Robin auf dem Empire State Building</a></li>
<li><a href="bilder/bild2.jpg"><img src="bilder/thumbnail2.jpg" alt="">
   Noël und Francis vor dem Taj Mahal</a></li>
<li><a href="bilder/bild3.jpg"><img src="bilder/thumbnail3.jpg" alt="">
   Isaak und Ahmet vor den Pyramiden von Gizeh</a></li>
<!-- ... viele weitere Links mit Thumbnails ... -->
</ul>
```

Beim Klick auf einen der Links soll nun das verlinkte Bild eingeblendet werden. Anstatt jedem `a`-Element einzeln einen Handler zuzuweisen, registrieren wir ihn beim gemeinsamen Vorfahrenelement`ul` mit der ID `bilderliste`:

```
document.getElementById("bilderliste").onclick = bilderlistenKlick;
```

In der angegebenen Handler-Funktion `bilderlistenKlick` findet nun die Überprüfung des Zielelementes statt.

```
funktion bilderlistenKlick (e) {
   // Vereinheitlichung von Event-Objekt und Zielelement
   var e = e || window.event;
   var target = e.target || e.srcElement;

   var elementName = target.nodeName,
      aElement = false;
   // Überprüfe, ob das Zielelement ein Link oder ein Bild im Link ist:
   if (elementName == "A") {
      // Falls ein Link geklickt wurde, speichere das Zielelement
      // in der Variable aElement:
      aElement = target;
   } else if (elementName == "IMG") {
      // Falls das Thumbnail-Bild geklickt wurde,
      // suche das zugehörige Link-Element:
      aElement = target.parentNode;
   }

   // Zeige das Vollbild, wenn das Zielelement
   // ein Link ist oder in einem Link liegt:
   if (aElement) {
      zeigeVollbild(aElement);
      // Unterdrücke die Standardaktion:
      return false;
   }

   // Andernfalls mache nichts.
}
```

In dieser Funktion wird das Zielelement des Ereignisses angesprochen und dessen Elementname überprüft. Wenn ein `a`-Element geklickt wurde, muss es sich um einen Link auf ein Bild handeln und das Vollbild soll eingeblendet werden.

Das alleine wäre bereits mit der Abfrage `if (target.nodeName == "A")` zu erledigen. Das Beispiel hat allerdings bewusst eine Schwierigkeit eingebaut, um Ihnen das Event-Bubbling und das Arbeiten mit dem Zielelement näher zu bringen: In den `a`-

Elementen liegen zusätzlich `img`-Elemente für die Thumbnails. Wenn der Anwender auf diese klickt, soll das Vollbild selbstverständlich ebenfalls eingeblendet werden. In dem Fall ist jedoch nicht der Link das Zielelement, sondern logischerweise das`img`-Element.

Aus diesem Grund muss die Abfrage erweitert werden: Handelt es sich um ein `a`-Element *oder* um ein Element, das direkt in einem `a`-Element liegt? Falls ein `img`-Element das Zielelement ist, steigen wir von diesem zu seinem `a`-Elternelement auf. Schließlich wird die Funktion `zeigeVollbild` mit dem gefundenen `a`-Elementobjekt als Parameter aufgerufen. Das Gerüst dieser Funktion sieht so aus:

```
function zeigeVollbild (aElement) {
   // Empfange das Elementobjekt als ersten Parameter und
   // lese dessen href-Attribut mit der Bild-Adresse aus:
   var bildAdresse = aElement.href;

   // Blende das Bild ein, auf das der Link zeigt.
   // (Die genaue Umsetzung ist an dieser Stelle ausgeklammert.)
}
```

Dieses Beispiel soll Ihnen die grundlegende Funktionsweise von Event-Delegation veranschaulichen:

1. Es gibt eine Handler-Funktion, die alle Ereignisse eines Types überwacht, welche von seinen Nachfahrenelementen aufsteigen.
2. Darin wird das Ereignis untersucht und insbesondere das Zielelement überprüft.
3. Wenn das Zielelement gewissen Kriterien entspricht (z.B. einem bestimmten Elementyp oder einer Klasse angehört), wird auf das Ereignis reagiert. Das kann in dieser Funktion erfolgen oder der Übersicht halber in einer anderen.

Wie Sie schon bei diesem einfachen Beispiel sehen, ist eine aufwändige Untersuchung des DOM-Elementenbaumes rund um das Zielelement nötig. Bei Event-Delegation stellt sich oft die Frage, ob das Zielelement in einem anderen Element enthalten ist, auf das gewisse Kriterien zutreffen. Eine allgemeinere und vielseitig einsetzbare Lösung werden Sie später noch kennenlernen.

Anwendungsbereiche von Event-Delegation

... Forumsposting

Capturing

Capturing (englisch für »Einfangen«) ist eine Phase beim Event-Fluss, die wir bereits kurz angesprochen haben. Der DOM-Event-Standard definiert **drei Phasen**, in denen ein Ereignis durch den DOM-Elementbaum wandert (Event-Fluss) und Handler auslöst:

1. **Capturing-Phase** (Absteigen zum Zielelement): Das Ereignis steigt vom obersten Dokument-Knoten im Elementenbaum hinab bis zum Zielelement des Ereignisses. Auf diesem Weg werden alle Handler ausgeführt, die für den Ereignistyp für die Capturing-Phase registriert wurden.
2. **Target-Phase** (Zielelement-Phase): Das Ereignis erreicht sein Zielelement und löst die betreffenden Handler aus, die dort für die Bubbling-Phase registriert wurden.
3. **Bubbling-Phase** (Aufsteigen vom Zielelement): Das Ereignis steigt ausgehend vom Zielelement wieder in der Element-Hierarchie auf. Es durchläuft alle Vorfahrenelemente und löst dort die relevanten Handler aus.

Alle bisher beschriebenen Modelle, ausgehend vom traditionellen über W3C DOM Events und demMicrosoft-Modell, haben Handler für die Bubbling-Phase registriert. Wie wir uns das Bubbling zunutze machen, haben wir bereits bei der Event-Delegation kennengelernt.

Capturing ist ein weiterer Ansatz, um Ereignisse effizienter zu überwachen: Wir können ein Event-Handler bei einem höherliegenden Element registrieren, um die Ereignisse zu überwachen, die bei vielen Nachfahrenelemente passieren.

Der Unterschied zwischen Bubbling und Capturing folgender: Nicht alle Ereignisse haben eine Bubbling-Phase, das heißt nicht alle Ereignisse steigen auf und lösen die entsprechenden Handler bei ihren Vorfahrenelementen aus. Das hat durchaus seinen Sinn, macht aber die beschriebene Event-Delegation unmöglich. Gäbe es das Event-Capturing nicht, wären Sie gezwungen, alle nicht aufsteigenden Ereignisse direkt bei ihren Zielelementen zu überwachen. Mithilfe des Event-Capturings können Sie auch solche

Ereignisse zentral überwachen – denn *jedes* Ereignis hat eine Capturing-Phase.

Event-Capturing ist nur unter Verwendung der standardisierten Methode `addEventListener`möglich. Das traditionelle Event-Handling mit seinem Schema `element.onevent = handlerfunktion` registriert den Handler immer für die Bubbling-Phase. Dasselbe gilt für das Microsoft-Modell mit `attachEvent`. Für Internet Explorer vor Version 9, welche `addEventListener`nicht unterstützen, müssen Sie daher gegebenenfalls eine Alternative ohne Event-Capturing bereitstellen.

Um Event-Handler für die Capturing-Phase zu registrieren, nutzen Sie wie gewohnt`addEventListener`, setzen jedoch den dritten Boolean-Parameter auf `true`:

```
document.addEventListener("focus", captureHandler, true);
```

Die Vorteile des Capturings liegen also darin, insbesondere nicht aufsteigende Ereignisse bei einem höherliegenden Element zu verarbeiten.

Folgende Ereignisse beispielsweise steigen nicht auf:

- `load`, z.B. bei Bildern, Objekten und Iframes
- `focus` und `blur`. (Als alternative aufsteigende Ereignisse gibt es allerdings `focusin` und`focusout`.)
- `mouseenter` und `mouseleave`. (Die Ereignisse `mouseover` und `mouseout` hingegen steigen auf.)
- `submit`

Da die Capturing-Phase die erst im Event-Fluss ist, ist es möglich, ein Ereignis schon in dieser Phase abzufangen. Ruft man in der Capturing-Phase die `stopPropation`-Methode des Event-Objektesauf, so wird der Fluss abgebrochen und die Ziel- und Bubbling-Phase fallen aus. Das Ereignis erreicht somit das Zielelement nicht.

JavaScript: Browserübergreifende Entwicklung

JavaScript und die Browser-Wirklichkeit

Wer schon etwas Erfahrung in der Webentwicklung gesammelt hat, kennt das Problem: Ein Großteil der Arbeit muss dafür investiert werden, Unterschiede und Fehler der verbreiteten Browser zu berücksichtigen. Besonders im Bereich CSS muss man mit Browserweichen arbeiten, um alle großen Browser zu einer halbwegs einheitlichen Darstellung zu bringen.

Wie sieht es im Bereich JavaScript aus? Zunächst einmal ähnlich schlecht. Manche Browser setzen die DOM-Standards nicht vollständig um und für zahlreiche Aufgaben existieren nur proprietäre Technike. Somit sind oftmals **mehrgleisige Scripte** nötig, die dieselbe Aufgabe je nach Browser auf eine unterschiedliche Weise lösen.

Dennoch gibt es keinen Grund, vor dem Browser-Durcheinander zu kapitulieren. Während es in CSS nur ungenaue Browserweichen gibt, ist es in JavaScript meistens möglich, gezielt die **Existenz der Objekte abzufragen**. Existiert ein Objekt und hat gegebenenfalls einen bestimmten Typ und Wert, kann damit gearbeitet werden. Andernfalls können Alternativlösungen greifen.

Verwendet man solche »**Fähigkeiten-Weichen**« anstelle von Browserweichen, die bloß über den Browsernamen Rückschlüsse ziehen, sind zuverlässige und zukunftsfähige Scripte möglich.

Zudem wurden mittlerweile die wichtigsten Unterschiede, Eigenheiten und Fehler der verbreiteten Browser in mühsamer Kleinarbeit dokumentiert. Dabei fielen fertige Script-Schnipsel und bewährte Verfahren ab, mit denen sich **Standardaufgaben browserübergreifend lösen** lassen. Schließlich ist in puncto JavaScript Bewegung in den Browsermarkt gekommen: Nach und nach werden Fehler bei der Umsetzung von Standards behoben, undokumentierte Techniken werden standardisiert und in die Browser eingebaut.

Dieser Ausblick soll optimistisch stimmen und dazu anregen, die Herausforderung der Browser-Wirklichkeit anzunehmen. Es ist jedoch auch ein hartes Stück Arbeit, sich in die browserübergreifende JavaScript-Entwicklung hineinzudenken.

Abwärtskompatibilität und Zukunftsfähigkeit

Was bedeutet eigentlich »browserübergreifende Entwicklung«? Üblicherweise wird darunter verstanden: Ein Script soll in allen Browsern den gewünschten Zweck erfüllen und dasselbe leisten. Von dieser einseitigen Vorstellung sollten Sie sich verabschieden, denn in vielen Fällen ist sie schlicht nicht umsetzbar!

Das realistische Ziel sollte vielmehr lautet, die vergangenen, gegenwärtigen und zukünftigen Browser **gemäß ihrer jeweiligen Fähigkeiten** zu bedienen. Dies bedeutet nicht, dass ihr Script auf allen Browsern und Browserversionen exakt denselben Effekt haben muss. Es ist nicht immer sinnvoll, technisch veralteten oder wenig verbreiteten Browser mit unverhältnismäßigem Aufwand dasselbe Ergebnis zu liefern.

Dennoch ist es durch geschickte Programmierung meistens möglich, alle relevanten Browsern eine funktionsfähige Seite zu präsentieren - auch wenn diese z.B. in älteren Browsern nicht ganz so ansehnlich und komfortabel bedienbar ist wie in weiter entwickelten Browsern.

Abwärtskompatibilität bedeutet, dass Ihr Script auf den Fall vorbereitet ist, dass gewisse JavaScript-Techniken nicht zur Verfügung steht oder von Seiten des Browsers fehlerhaft umgesetzt sind.*Zukunftsfähigkeit* bedeutet, dass Sie durchaus neue und noch nicht breit unterstützte Techniken verwenden können. Voraussetzung ist jeweils, dass sie keine Techniken stillschweigend voraussetzen, sondern immer prüfen, ob die benötigten Objekte existieren und die Teilanweisungen ihres Programmes die erwarteten Ergebnisse liefern.

Fallunterscheidungen und Vereinheitlichungen

Das Grundelement der browserübergreifenden Programmierung ist der mehrgleisiger Ablauf. Scripte nutzen immer wieder eine solche Struktur:

```
Wenn die nötige Technik zur Verfügung steht,
Dann:
  Löse das Problem auf die eine Weise
Andernfalls:
    Wenn die Alternativtechnik zur Verfügung steht,
    Dann:
      Löse die Aufgabe auf eine andere Weise
```

Diese Fallunterscheidungen werden in JavaScript üblicherweise mit bedingten Anweisungen umgesetzt: `if (Bedingung) { Anweisungen } else { Anweisungen }`

Entscheidend ist, genau die Unterschiede zu kennen und *punktuell* solche Abzweigungen einzubauen. Nicht das ganze Script sollte eine solche Struktur haben, es sollte sich dieser verzweigten Abläufe nur dort bedienen, *wo es nötig ist*. Auf diese Weise sparen Sie sich doppelte Arbeit.

Wo Unterschiede auftreten, sollten Sie mit solchen Abfragen möglichst für **Vereinheitlichung**sorgen. Darauffolgende Anweisungen müssen sich dann um die Unterschiede keine Gedanken mehr machen. Bis zur nächsten punktuellen Fallunterscheidung folgt wieder Code, der von allen Browsern verstanden wird.

Nehmen wir ein Beispiel aus dem : Eine Handler-Funktion bekommt das in manchen Browsern als Parameter übergeben. Im

Internet Explorer ist es jedoch nur über `window.event` zugänglich. Über das Event-Objekt hat man Zugriff auf das , bei dem das Ereignis passiert ist, welches die Handler-Funktion gerade verarbeitet. Auch in dem Punkt unterscheiden sich die Browser: Das Element ist entweder in der Eigenschaft `target` (W3C-konforme Browser) oder `srcElement` (Internet Explorer) gespeichert. Nun könnte man folgendermaßen vorgehen:

```
function handlerFunktion (eventObjekt) {
  if (eventObjekt) {
    alert("Element, an dem das Ereignis passierte: " + eventObjekt.target.nodeName);
  } else if (window.event) {
    alert("Element, an dem das Ereignis passierte: " + window.event.srcElement.nodeName);
  }
}
```

Allerdings ist dieser Code stellenweise redundant und setzt stillschweigend voraus, dass die Übergabe des Event-Objektes als Parameter und `target` sowie `window.event` und `srcElement`notwendigerweise zusammengehören - das ist zwar bei diesem Beispiel kein Problem, aber in anderen Fällen kann eine solche Annahme Probleme bringen.

Anstatt direkt browserspezifischen Code zu schreiben, nutzen wir **punktuelle Vereinheitlichung**, die gleiche Voraussetzungen für das Script schafft. Nachdem die Browserunterschiede eingeebnet wurden, können wir die Aufgabe, die wir eigentlich lösen wollen, viel einfacher umsetzen. Vor allem wird der Code übersichtlicher und besser strukturiert: Die Bereiche, die sich nur den Browserproblemen widmen, sind getrennt von denen, die die eigentliche Aufgabe lösen.

```
function handlerFunktion (eventObjekt) {

  // Vereinheitliche den Zugriff auf das Event-Objekt:
  if (!eventObjekt) {
    eventObjekt = window.event;
  }
  // Das Objekt ist jetzt browserübergreifend in der Variable eventObj gespeichert.

  // Vereinheitliche den Zugriff auf das Ziel-Element:
  var target;
  if (eventObjekt.target) {
    target = eventObjekt.target;
  } else if (eventObjekt.srcElement) {
    target = eventObjekt.srcElement;
  }
  // Das Objekt ist nun browserübergreifend in der Variable target gespeichert.

  // Nach den Vereinheitlichungen folgt die eigentliche Umsetzung.
  // Stelle das Gewünschte mit dem Ziel-Element an:
  alert("Element, an dem das Ereignis passierte: " + target.nodeName);

}
```

Diese Umsetzung mag zunächst länger und umständlicher scheinen. Das liegt jedoch bloß an der ausführlichen Schreibweise. Eine mögliche Kurzschreibweise könnte so aussehen:

```
function handlerFunktion (e) {
  e = e || window.event;
  var target = e.target || e.srcElement;
  alert("Element, an dem das Ereignis passierte: " + target.nodeName);
}
```

Mit dem `||`-Operator wird hier die Fallunterscheidung umgesetzt. Was kryptisch aussehen mag, hat bloß folgenden Effekt: Wenn der Parameter `e` gesetzt ist, nehme diesen als Event-Objekt, andernfalls versuche es mit `window.event`.

Dasselbe Schema wird in der nächsten Zeile angewendet: Wenn die Eigenschaft `target` des Event-Objektes gesetzt ist, so speichere diese in der Variable `target`, andernfalls verwende die Eigenschaft `srcElement`.

Todo: Erklärung des ||-Operators in den Sprachgrundlagen verlinken

Das Beispiel lässt außen vor, dass Browser denkbar sind, die weder die eine noch die andere Vorgehensweise unterstützen. Oder die Handler-Funktion wird aus Versehen außerhalb einer Ereignis-Verarbeitung aufgerufen. Es ist nicht immer nötig, diese denkbaren, aber unrealistischen Fälle abzudecken. Sie können es dennoch tun, indem Sie abfragen, ob die Variablen `e` und `target` nach der Vereinheitlichung korrekt gesetzt sind. Wenn dies nicht gegeben ist, kann die Funktionsausführung z.B. durch die Anweisung `return;` abgebrochen werden.

```
e = e || window.event;
if (!e) {
  // Kein Zugriff auf das Event-Objekt möglich, breche Funktion ab,
  // um einen Scriptabbruch zu umgehen.
  return;
}
```

Beachten Sie, dass diese Schreibweise zum Überprüfen der Variable `e` nur Sinn ergibt, wenn es sich um einen Funktionsparameter handelt, der entweder den Wert `undefined` hat (falls kein Parameter übergeben wurde) oder eben das Event-Objekt enhält. Beim Umgang mit anderen Objekttypen sähe die Abfrage anders aus.

Fähigkeitenerkennung statt Browsererkennung

Lange Zeit bedienten sich browserübergreifende JavaScripte einer sogenannten Browsererkennung. Anstatt in Erfahrung zu bringen, mit welchen konkreten Unterschieden das Script konfrontiert ist, fragte man kurzerhand den **Browsernamen** ab. Die Struktur einer solchen Browserweiche sah etwa so aus:

```
Wenn der Browser den Namen »Internet Explorer« hat,
Dann:
  Löse die Aufgabe auf die die IE-typische Weise
Andernfalls:
  Löse die Aufgabe auf die Netscape-typische Weise
```

Umgesetzt wurden solche Abfrage mit dem JavaScript-Objekt `window.navigator`, das verschiedene Informationen über den Browser liefert, der das JavaScript ausführt.

In dem obigen Beispiel werden allein die Browser Internet Explorer und Netscape Navigator berücksichtigt, die sich Ende der 1990er Jahre gegenüberstanden und den Browsermarkt beherrschten. Diese Vorgehensweise ging so lange gut, wie nur diese beiden Browser verbreitet waren und sich deren Versionen gleich verhielten. Diese Situation war jedoch höchstens für eine kurze Zeit gegeben - danach funktionierten solche Scripte nicht mehr zuverlässig.

Alternativ zur Abfrage des Browsernamens wurden **zentrale Objekte** zur Browsererkennung verwendet:

```
Wenn das Objekt document.all existiert,
Dann:
  Nehme an, es ist ein Internet Explorer und löse die Aufgabe
  auf die die IE-typische Weise
Andernfalls:
  Wenn das Objekt document.layers existiert,
  Dann:
    Nehme an, es ist ein Netscape Navigator und löse die Aufgabe
    auf die Netscape-typische Weise
```

Solche Objektabfragen sind nicht völlig abwegig, denn die abgefragten Objekte `document.all`und `document.layers` wurden bei der Umsetzung meistens auch verwendet. Wenn jedoch von der Existenz *eines* Objektes wird stillschweigend auf die Existenz vieler anderer Browserfähigkeiten geschlossen wird, handelt es sich um

eine versteckte Browserabfrage.

Eine Browserweiche geht davon aus, dass der Browser eine Reihe von Techniken unterstützt, nur weil er einen bestimmten Namen trägt oder ein zentrales Objekt existiert. Zum einen können damit immer nur die derzeit bekannten Browser in ihren aktuellen Versionen berücksichtigt werden.

Zum anderen halten sich viele Browser an herstellerunabhängige Standards, die sie mehr oder weniger korrekt umsetzen. Aber auch zahlreiche proprietäre, das heißt ursprünglich browserspezifische Erfindungen sind nicht mehr auf einen Browser begrenzt. Andere Hersteller haben sie ebenfalls übernommen. »Browserabfragen« sind schon aus diesem Grund nicht zuverlässig und zukunftsfähig.

Beispielsweise wird das längst veraltete Objekt `document.all` immer noch zur Erkennung des Internet Explorers verwendet. Das ist unklug, denn oftmals gibt es andere Browser, die die verwendeten Fähigkeiten beherrschen, jedoch nicht `document.all`.

Browserweichen können daher prinzipiell nicht alle Fälle angemessen berücksichtigen. Sie sollten sie möglichst vermeiden und stattdessen abfragen, ob der jeweilige Browser die *Fähigkeiten und Teiltechniken unterstützt*, die sie *tatsächlich in ihrem Script verwenden*.

Objektabfragen

Objekte und Methoden abfragen

Die einzelnen Fähigkeiten eines Browsers drücken sich meist darin aus, dass bestimmte vordefinierte Objekte, Eigenschaften bzw. Methoden existieren. In manchen Fällen müssen Sie zusätzlich prüfen, ob die Eigenschaft auch einen bestimmten Typ oder Wert hat.

Wenn wir eine bedingte Anweisung mit `if (...) {...}` notieren, so wird die Bedingung (der Ausdruck zwischen den runden Klammern) letztlich in einen Boolean-Wert, also `true` oder `false`umgewandelt. Allgemeine Objekte vom Typ `Object` ergeben bei der Umwandlung in dem Typ `Boolean`den Wert `true`.

Das bedeutet, Sie können einfach `if (objekt.unterobjekt) { ... }` notieren, um die Existenz des Unterobjektes abzufragen. Für Funktionsobjekte gilt dasselbe, also notieren wir `if (objekt.methode) { ... }`. Die Schreibweise `objekt.unterobjekt` bzw. `objekt.methode` ist dabei entscheidend. Nur in besonderen Fällen ist dies durch `if (objekt)` bzw. `if (methode)`ersetzbar.

Das folgende Beispiel veranschaulicht die Existenzabfrage von Objekten und Methoden. Die Funktion bringt den aktuell markierten Text im Dokument in Erfahrung und gibt diesen in einem Meldungsfenster aus.

```
function selektierterText () {
  var text = "";

  if (window.getSelection) {
    text = window.getSelection();
  } else if (document.getSelection) {
    text = document.getSelection();
  } else if (document.selection && document.selection.createRange) {
    text = document.selection.createRange().text;
  } else {
    return;
  }

  alert(text);
}
```

Es existieren für diese Aufgabenstellung **drei Lösungsweisen**, die in unterschiedlichen Browsern zum Ziel führen. Das Beispiel demonstriert daher eine dreiteilige Fallunterscheidung mit verschachtelten`if-else`-Anweisungen.

Je nachdem, welche Objekte bzw. Methoden existieren, werden diese verwenden. Kennt der JavaScript-Interpreter keines dieser Objekte, wird die Funktion vorzeitig beendet. Die verschachtelten Fallunterscheidung in verständlicher Sprache:

```
Existiert die Methode window.getSelection?
Falls ja:
  benutze diese Methode.
Falls nein:
  Existiert die Methode document.getSelection?
  Falls ja:
    Benutze diese Methode.
  Falls nein:
    Existiert das Objekt document.selection und hat es eine Methode createRange?
      Falls ja:
        Benutze diese Methode.
      Falls nein:
        Breche ab.
```

Wie Sie sehen, kommt dieses Beispiel ganz ohne Browserabfragen aus und ist doch für verschiedene Browser ausgelegt.

Andere Typen abfragen

Wenn Sie die Existenz von Objekten anderer Typen prüfen wollen, müssen Sie gegebenenfalls anders vorgehen.

Zahlen (Typ `Number`) und Zeichenketten (Typ `String`) können Sie im Prinzip zwar auch mit `if (objekt.numberEigenschaft)` bzw. `if (objekt.stringEigenschaft)` abfragen. Die Bedingung in den Klammern wird aber wie gesagt in ein `Boolean`-Wert umgewandelt. Leere Zeichenketten (`""`) und gewisse Zahlenwerte (z.B. `0`) ergeben bei dieser Umwandlung `false`. Diese Umwandlungsregeln können Verwirrung stiften: Manchmal ist `0` ein gültiger Wert und Sie können damit arbeiten, in anderen Fällen weist er darauf hin, dass der Browser die von Ihnen benötigte Fähigkeit nicht hat und Sie die abgefragte Eigenschaft nicht verwenden können.

ToDo: Link auf "Was ist alles true/false?"

Wenn Sie mit solchen Typen umgehen, sollten Sie daher auf den Operator `typeof` ausweichen. Dieser gibt den Typ einer Eigenschaft als String zurück. In den beschriebenen Fällen wäre das`"string"` bzw. `"number"`). Ein Beispiel ist die Abfrage der Existenz von `window.innerHeight`:

```
if (typeof window.innerHeight == "number") {
  alert("Der Anzeigebereich des Browserfensters ist " +
    window.innerHeight + " Pixel breit!");
}
```

Da manche Browser, insbesondere der Internet Explorer beim Einsatz `typeof` den Typen nicht immer korrekt wiedergeben, hat sich eingebürgert, bei Existenzabfragen bloß zu prüfen, ob `typeof`nicht `"undefined"` liefert:

```
if (typeof window.innerHeight != "undefined") {
  alert("Der Anzeigebereich des Browserfensters ist " +
    window.innerHeight + " Pixel breit!");
}
```

ToDo: siehe den weiterführenden Artikel Objektabfragen und Fallunterscheidungen in JavaScript

Browsererkennung in Sonderfällen

Eine Fähigkeitenerkennung mit Abfragen der verwendeten Objekte ist in den meisten Fällen möglich und vorzuziehen. Wenn Sie diese Vorgehensweise gewohnt sind, werden Sie bei neuen Aufgaben schnell einen Weg finden, wie sie die Browserunterschiede in Ihrem Script zuverlässig prüfen und entsprechende Fallunterscheidungen einbauen können. Es sei allerdings nicht verschwiegen, dass sich manche Browsereigenheiten und -fehler nicht so einfach erkennen lassen. Diese Fälle sind glücklicherweise äußerst selten.

Ein Beispiel: Sie prüfen, ob eine Methode existiert und rufen Sie auf, falls sie existiert. Wenn ein Browser die Methode nun kennt, aber falsch implementiert hat, sodass sie nicht das beabsichtigte tut, können Sie dies nicht immer erkennen. Wenn auch der

Rückgabewert in Ordnung scheint, können Sie den Browserfehler schwer feststellen.

In diesen Ausnahmefällen bleibt oft nichts anderes als eine Browserabfrage übrig. Wenn Sie eine solche einbauen, müssen Sie darauf achten, dass das Script nicht unter unerwarteten Umständen fehlerhaft arbeitet. Beispielweise kann der Browserfehler, den Sie umgehen wollen, schon in der nächsten Version behoben sein. Dann kann es sein, dass Ihr Script abbricht. Eine Regel lautet daher, nur bekannte Browserversionen anzusprechen. Doch auch diese Vorgehensweise hat ihren Nachteil: Erscheint eine neue Version, die den Fehler immer noch beinhaltet, müssen Sie das Script zeitnah aktualisieren und die Erkennung anpassen. Wie Sie sehen, handelt Sie sich mit Browserabfragen schnell Probleme ein.

Eine allgemeine Browsererkennung ist über das Objekt `window.navigator` und dessen Eigenschaften möglich. Ein Fertigscript, das Ihnen Browsername und -version zur Verfügung stellt, finden Sie unter Browser detect [en]. Seien Sie sich jedoch darüber im klaren, dass diese `navigator`-Eigenschaften nicht verfügbar oder manipuliert sein können - eine Erkennung über `navigator` ist nicht hundertprozentig zuverlässig!

Für spezifische Browser gibt es zuverlässigere Erkennungsmethoden:

Opera bietet das Objekt `window.opera` an, welches auch nur im Opera existiert. Dieses können Sie einfach mit `if (window.opera) {...}` abfragen.

Wenn Sie den **Internet Explorer** in einem Script erkennen wollen, so können Sie alternativ zu*Conditional Compilation* (engl. bedingte Ausführung) greifen. Dabei handelt es sich um eine Microsoft-eigene Zusatzsprache, die Sie in den JavaScript-Code einfügen können. Im Grunde handelt es sich um JavaScript-Kommentare nach dem Schema `/*@cc_on ... @*/`. Während die anderen Browser sie ignorieren, erkennt der Internet Explorer `@cc_on` eine Anweisung. In diesem Fall: Schalte Conditional Compilation (CC) an. Das bedeutet, dass der Internet Explorer den Text zwischen `@cc_on` und `@` als JavaScript-Code behandelt und ausführt, obwohl diese sich in einem Kommentar befindet. Lange Rede, kurzer Sinn: Auf diese Weise können Sie JavaScript-Code notieren, der garantiert nur vom Internet Explorer ausgeführt wird - denn vor allen anderen Browser ist er in einem Kommentar versteckt. Ein Beispiel:

```
if (/*@cc_on ! @*/ false) {
  window.alert("Dies wird nur im Internet Explorer ausgeführt!");
}
```

Alle Browser außer dem Internet Explorer führen den Code `if (false) {...}` aus, nur im Internet Explorer wird mittels speziellem Kommentar ein Negationsoperator (das Ausrufezeichen) eingefügt. Der IE führt also `if (!false) {...}` und damit die Anweisungen im `if`-Block aus.

JavaScript: Zusammenarbeit mit CSS, Darstellung von Dokumenten steuern

Einleitung

Mit JavaScript können Sie die Darstellung des Dokuments dynamisch ändern, während es im Browser angezeigt wird. Dies ist ein wesentlicher Bestandteil der Interaktivität, die Sie einem Dokument mittels Javascript hinzufügen können. Die Möglichkeiten sind vielfältig: Beispielsweise können Sie als Reaktion auf eine Benutzereingabe gewisse Elemente ein- und ausblenden. Es sind aber auch – mit entsprechendem Aufwand – visuelle Effekte und komplexe Animationen möglich.

Die Programmiersprache JavaScript besitzt keine eigenen Techniken, um die Gestaltung einer Webseite zu beeinflussen. Vielmehr besitzt JavaScript eine Schnittstelle zur Formatierungssprache Cascading Stylesheets (CSS). Mittels JavaScript können Sie also sämtliche Formatierungen vornehmen, die CSS möglich macht. Daher sollten Sie die Grundlagen von CSS bereits beherrschen, bevor Sie Dokumente mittels JavaScript umformatieren.

Das dynamische Ändern der *Darstellung* bildet einen großen Komplex in der JavaScript-Programmierung - ein anderer ist das dynamische Ändern der *Inhalte* über das Document Object Model (DOM). Über das DOM können Sie Elemente hinzufügen oder löschen, Attributwerte setzen und Textinhalte einfügen oder verändern. In der Praxis gehen diese beiden Aufgaben - den Inhalt und dessen Darstellung modifizieren - oft miteinander einher.

Trennung von Layout-Regeln und JavaScript-Logik

Bevor Sie die verschiedenen Möglichkeiten kennenlernen, wie Sie ein Element CSS-Formatierungen mithilfe von JavaScript verändern können, sollten Sie sich die Konzepte des Unobtrusive JavaScript in Erinnerung rufen.

Wenn Sie bereits mit CSS fortgeschritten sind und einige Layouts mit CSS umgesetzt haben, sollten Sie die Aufgaben der Webtechniken kennen und deren sinnvolle Anwendung bereits beherrschen:

- Für die *Strukturierung der Inhalte* ist HTML zuständig. Sie wählen möglichst bedeutungsvolle HTML-Elemente und für die Feinstrukturierung vergeben Sie Klassen und IDs.
- Die *Präsentation* hingegen sollte nicht mittels HTML beeinflusst werden, sondern mit ausgelagerten Stylesheets. Diese sprechen gezielt Elemente im Dokument an und formatieren sie mit CSS-Eigenschaften. Idealerweise bleiben die HTML-Strukturen übersichtlich, IDs und Klassen sind sparsam gesetzt und aussagekräftig, sodass eindeutige Angriffspunkte für CSS-Regeln existieren.

Diese Arbeitsweise hat bringt Ihnen enorme Vorteile bei der Webseiten-Entwicklung. Inhalte und Präsentation können unabhängig voneinander schnell geändert werden, mit wenig Aufwand kann die gewünschte Präsentation erzielt werden.

Wenn nun die dritte Technik – JavaScript – hinzutritt, sollten Sie dieses Modell konsequent fortführen. Orientieren Sie sich an folgenden Faustregeln:

- Definieren Sie die Formatierungsregeln im zentralen Stylesheet, nicht im JavaScript. Trennen sie den CSS-Anweisungen vom JavaScript-Code.
- Sorgen Sie im JavaScript dafür, dass diese Formatierungsregeln angewendet werden – beispielsweise indem Sie einem Element dynamisch eine Klasse hinzufügen. Durch diese Änderung der Klasse kann eine Regel im Stylesheet greifen, deren Selektor die soeben gesetzte Klasse enthält.

Sie können nicht nur ausgelagerte Stylesheet-Regeln auf ein Element anwenden, sondern auch direkt gewisse CSS-Eigenschaften von einzelnen Elementen ändern können. Dies entspricht dem `style`-Attribut in HTML. Diese Vermischung von HTML und CSS bzw. JavaScript und CSS sollten Sie möglichst vermeiden. Diese Direktformatierung ergibt nur in Sonderfällen Sinn, deshalb sollten Sie sich zunächst mit der besagten Arbeitsweise vertraut machen.

Stylesheet-Regeln auf ein Element anwenden

Eine einfache Methode, um die Darstellung von Elementen per JavaScript zu ändern, ist das**Setzen einer Klasse**. Die Formatierungen für diese Klasse wird im Stylesheet untergebracht. Damit wird der empfohlenen Trennung vom JavaScript-Code Genüge getan.

Betrachten wir als Beispiel ein JavaScript, dass die Eingaben eines Formulars überprüft. Beim Absenden des Formulars wird eine Handler-Funktion für das Ereignis `submit` aktiv. Diese Funktion soll fehlerhafte Formularfelder rot markieren.

Nehmen wir an, die `input`-Eingabefelder sind standardmäßig im Stylesheet so formatiert:

```
input {
    padding: 4px 6px;
    border: 1px solid #555;
    background-color: #fafafa;
}
```

Im Stylesheet wird nun eine Regel definiert mit den Eigenschaften für fehlerhafte Felder. Wir nutzen dazu einen Selektor mit dem Elementnamen `input` kombiniert mit der Klasse `fehlerhaft`:

```
input.fehlerhaft {
   border-color: red;
   background-color: #fff8f5;
}
```

Am Anfang trifft der Selektor `input.fehlerhaft` auf kein Feld im Dokument zu – solange nicht ein Feld die Klasse `fehlerhaft` besitzt. Um ein Eingabefeld umzuformatieren und die Regel anzuwenden, vergeben wir dynamisch diese Klasse an das gewünschte `input`-Element.

Die Klassen eines Elements sind in JavaScript über die Eigenschaft `className` des entsprechenden Elementobjektes zugänglich. Sie können diese Eigenschaft sowohl auslesen als auch ihr einen neuen Wert zuweisen. Das vereinfachte Schema zum Setzen einer Klasse lautet:

```
element.className = 'klassenname';
```

In der beispielhaften Formularüberprüfung kann das Setzen der Klasse folgendermaßen aussehen: Wir definieren eine Funktion `formularÜberprüfung`, die als Handler für das Ereignis `submit`registriert wird (siehe Ereignisverarbeitung). In dieser Funktion wird das zu überprüfende Formularfeld über das DOM mittels `document.getElementById` herausgesucht. Ist der Wert des Feldes leer, wird eine Meldung ausgegeben und das Feld bekommt die Klasse `fehlerhaft`.

```
function formularÜberprüfung () {
    // Spreche das Formularfeld über das DOM an und
    // speichere es in eine Variable zwischen:
    var element = document.getElementById("kontaktformular-name");
    // Prüfe den Feldwert:
    if (element.value == "") {
        // Zeige im Fehlerfall ein Hinweisfenster:
        window.alert("Bitte geben Sie Ihren Namen an.");
        // Weise dem Element die Klasse »fehlerhaft« zu:
        element.className = 'fehlerhaft';
        // Setze den Fokus auf das Feld:
        element.focus();
        // Verhindere das Absenden des Formulars
        // (unterdrücke die Standardaktion des Ereignisses):
        return false;
    }
}

function init () {
    // Starte die Ereignis-Überwachung mittels
    // traditionellem Event-Handling
    document.getElementById("kontaktformular").onsubmit = formularÜberprüfung;
}
```

```
window.onload = init;
```

Der zugehörige HTML mit den nötigen IDs könnte so aussehen:

```
<form action="…" method="post" id="kontaktformular">
    <p><label>
        Ihr Name:
        <input type="text" name="name" id="kontaktformular-name">
    </label></p>
    … weitere Felder …
    <p><input type="submit" value="Absenden"></p>
</form>
```

Der Clou dieser Vorgehensweise ist, dass Sie mit dem Setzen der Klasse an einem Element nur eine minimale JavaScript-Änderung vornehmen. Diese Änderung führt dazu, dass eine Regel aus dem Stylesheet plötzlich auf bestimmte Elemente greift – der Browser wendet daraufhin automatisch die definierten Formatierungen an.

Über dieses Modell können Sie auch komplexere Aufgabenstellungen lösen, denn Ihnen stehen alle Möglichkeiten von CSS-Selektoren zu Verfügung. Beispielsweise können Sie mittels Nachfahrenselektoren Elemente formatieren, die unterhalb des mit der Klasse markierten Elements liegen. So können Sie durch die Änderung der Klasse gleich mehrere enthaltene, im DOM-Baum unterhalb liegende Elemente formatieren, ohne diese einzeln anzusprechen.

TODO: Beispiel dazu. Mit Nachfahrenselektoren größere Umformatierungen vornehmen, ohne alle Elemente einzeln anzusprechen

Komfortables Hinzufügen, Löschen und Abfragen von Klassen

Die oben vorgestellte Methode zum Setzen der Klasse ist stark vereinfacht und hat verschiedene Nachteile. Ein HTML-Element kann nämlich mehreren Klassen angehören. Die JavaScript-Eigenschaft`className` enthält dann eine Liste von Klassen, die durch Leerzeichen getrennt werden.

Beispielsweise kann im HTML `<input class="klasse1 klasse2">` notiert sein. Wenn Sie nun mittels JavaScript eine dritte Klasse hinzufügen wollen, so können Sie nicht einfach`element.className = "klasse3"` notieren, denn dies würde die ersten beiden Klassen löschen. Dasselbe gilt für das Entfernen einer Klasse: Wenn Sie einfach den Attributwert mit`element.className = ""` leeren, dann löschen Sie alle Klassenzugehörigkeiten.

Aus diesem Grund sollten Sie nicht direkt mit der `className`-Eigenschaft arbeten, sondern für diese Aufgaben Helferfunktionen verwenden, die mehrere Klassen berücksichtigen. Die meisten Allround-Bibliotheken bieten entsprechenden Funktionen an. Üblicherweise tragen sie folgende Namen:

- `addClass` zum Hinzufügen einer Klasse,
- `removeClass` zum Löschen einer Klasse,
- `hasClass` zur Überprüfung auf Zugehörigkeit zu einer Klasse. (Die Funktion gibt einen Boolean-Wert zurück.)
- Hinzu kommt oftmals `toggleClass`, welche eine Klasse hinzufügt oder löscht je nachdem, ob sie bereits gesetzt ist.

Falls Sie keine Fertigbibliothek nutzen, können Sie diese Helferfunktionen dennoch in ihre Scripte aufnehmen. Eine mögliche Umsetzung als lose globale Funktionen sieht folgendermaßen aus:

```
function addClass (element, className) {
    if (!hasClass(element, className)) {
        if (element.className) {
            element.className += " " + className;
        } else {
            element.className = className;
        }
```

```
    }
}

function removeClass (element, className) {
var regexp = addClass[className];
    if (!regexp) {
        regexp = addClass[className] = new RegExp("(^|\\s)" + className + "(\\s|$)");
    }
    element.className = element.className.replace(regexp, "$2");
}

function hasClass (element, className) {
    var regexp = addClass[className];
    if (!regexp) {
        regexp = addClass[className] = new RegExp("(^|\\s)" + className + "(\\s|$)");
    }
    return regexp.test(element.className);
}

function toggleClass (element, className) {
    if (element.hasClass(className)) {
        element.removeClass(className);
    } else {
        element.addClass(className);
    }
}
```

Alle Funktionen erwarten jeweils zwei Parameter, nämlich das Elementobjekt und den gewünschten Klassennamen als String. Folgende Beispiele sollen die Anwendung illustrieren:

```
// Element ansprechen und Elementobjekt in einer Variable zwischenspeichern:
var element = document.getElementById("beispielID");
// Klasse hinzufügen:
addClass(element, "beispielklasse");
// Klasse löschen:
removeClass(element, "beispielklasse");
// Klasse an- und ausschalten je nach vorherigem Status:
toggleClass(element, "beispielklasse");
// Vorhandensein einer Klasse prüfen:
if (hasClass(element, "beispielklasse")) {
    window.alert("Klasse gefunden.");
} else {
    window.alert("Klasse nicht gefunden.");
}
```

Mit diesen Funktionen in Ihrem JavaScript-Werkzeugkasten können Sie das Zusammenspiel von JavaScript-Interaktivität und Stylesheet-Formatierungen komfortabel und übersichtlich meistern.

Direktformatierung über das `style`-Objekt

Inline-Styles in HTML

Um direkt einzelne HTML-Elemente mit CSS zu formatieren, existiert das `style`-Attribut, welches eine Liste von Eigenschafts-Wert-Zuweisungen enthält. Ein HTML-Beispiel:

```
<p style="color: red; background-color: yellow; font-weight: bold;">Fehler!</p>
```

Gegenüber dem Einsatz von zentralen Formaten in Stylesheets sind diese sogenannten *Inline-Styles* (eingebettete Formatierungen) ineffektiv und führen zu zu einer Vermischung von HTML und CSS, die die Wartbarkeit des Dokuments verschlechtert. Sie sollten Sie daher nur in Ausnahmefällen einsetzen, auf die wir später noch zu sprechen kommen.

Das `style`-Objekt als Schnittstelle zu Inline-Styles

JavaScript bietet eine Schnittstelle zu diesem `style`-Attribut: Das `style`-Objekt bei jedem Elementobjekt. Das `style`-Objekt hat für jede mögliche CSS-Eigenschaft eine entsprechende les- und schreibbare Objekteigenschaft. Zu der CSS-Eigenschaft `color` existiert also eine Objekteigenschaft`element.style.color` vom Type String.

CSS-Eigenschaftsnamen mit Bindestrichen, wie z.B. `background-color`, können nicht unverändert als JavaScript-Eigenschaftsnamen übernommen werden. Deshalb werden sie im sogenannten *Camel-Case* (Groß- und Kleinschreibung im Kamel-Stil) notiert: Der Bindestrich fällt weg, dafür wird der darauf folgende Buchstabe zu einem Großbuchstaben. Aus `background-color` wird also `backgroundColor`, aus `border-left-width` wird `borderLeftWidth` und so weiter. Die Großbuchstaben in der Wortmitte werden mit Höcker eines Kamels verglichen.

Folgendes Beispiel veranschaulicht das Setzen der Hintergrundfarbe eines Elements auf rot:

```
document.getElementById("beispielID").style.backgroundColor = "red";
```

Als Werte müssen Sie stets Strings angeben genau in der Form, wie sie in CSS spezifiziert sind. Das gilt auch für Zahlenwerte, die eine Einheit erfordern:

```
element.style.marginTop = 15; // Falsch!
element.style.marginTop = "15px"; // Richtig
```

Sonderfälle bei der Umsetzung von CSS- in JavaScript-Eigenschaftsnamen

Abweichungen vom besagten Schema

`cssFloat` vs. `styleFloat`

Sinnvoller Einsatz des `style`-Objektes

Das Setzen von CSS-Formatierungen direkt über das `style`-Objekt ist zwar einfach. Doch Sie diese Präsentationsregeln wie gesagt nicht im JavaScript, sondern sie z.B. in einer Klasse im Stylesheet unterbringen. Nur in manchen Fällen ist die Verwendung von Inline-Styles notwendig: Wenn der Eigenschaftswert nicht fest steht, sondern erst im JavaScript berechnet wird. Das ist der Fall z.B. bei Animationen oder bei einer Positionierung abhängig von der Mauszeiger-Position wie beim Drag and Drop.

`style` ist nicht zum Auslesen der gegenwärtigen Eigenschaftswerte geeignet

Das `style`-Objekt wird immer wieder missverstanden: **Sie können über das `style`-Objekt nichtden aktuellen, berechneten Wert einer CSS-Eigenschaft auslesen.** Sie können damit lediglich Inline-Styles setzen und die bereits gesetzten auslesen.

Die besagten Objekteigenschaften (`.style.`*`cssEigenschaft`*) sind allesamt **leer**, wenn sie nicht im betreffenden HTML-Element über ein `style`-Attribut oder wie beschrieben mit JavaScript gesetzt wurden. Folgendes Beispiel verdeutlicht dies:

```
<p id="ohne-inline-styles">Element ohne Inline-Styles</p>
<p id="mit-inline-styles" style="color: red">Element mit Inline-Styles</p>
```

```
// Gibt einen leeren String aus:
window.alert(
    document.getElementById("ohne-inline-styles").style.backgroundColor
);
// Gibt »red« aus, weil Inline-Style gesetzt wurde:
window.alert(
```

```
    document.getElementById("mit-inline-styles").style.backgroundColor
);
```

CSS-Eigenschaften auslesen

Über das `style`-Objekt besteht wie gesagt kein Zugriff auf den aktuellen CSS-Eigenschaftswert eines Elements, sofern kein entsprechender Inline-Style gesetzt wurde. Dies ist meistens der Fall, denn die Formatierungen gehen üblicherweise auf zentrale Formatierungen in ausgelagerten Stylesheets und auf die Standarformatierungen des Browsers zurück.

Das Auslesen des gegenwärtigen Werts von CSS-Eigenschaften eines Elements gestaltet sich als schwierig. Wenn man in Erfahrung bringen will, welche tatsächliche Textfarbe oder welche Pixel-Breite ein Element hat, dann ist nach den sogenannten *berechneten Werten* (englisch *computed values*) gefragt, wie sie in der CSS-Fachsprache genannt werden.

`getComputedStyle` aus W3C DOM CSS

Der W3C-Standard, der die Schnittstelle zwischen JavaScript und CSS und damit der Darstellung eines Dokuments festlegt, kennt zu diesem Zweck die Methode `window.getComputedStyle()`. Sie erwartet ein Elementobjekt als ersten Parameter und einen String mit einem CSS-Pseudo-Element als zweiten Parameter (beispielsweise `"after"`, `"before"`, `"first-line"` oder `"first-letter"`). Wenn man nicht die Formatierung des Pseudo-Elements abfragen will, übergibt man schlichtweg `null`als zweiten Parameter.

`getComputedStyle` gibt ein Objekt zurück, das genauso aufgebaut ist wie das bereits besprochene `element.style`-Objekt. Es enthält für jede CSS-Eigenschaft eine entsprechende Objekteigenschaft mit dem aktuellen berechneten Wert.

```
var element = document.getElementById('beispielID');
var computedStyle = window.getComputedStyle(element, null);
window.alert("Textfarbe: " + computedStyle.color);
window.alert("Elementbreite: " + computedStyle.width);
```

Die berechneten Werte (*computed values*), die `getComputedStyle` zurückgibt, sind nicht in jedem Fall identisch mit den Werten, die Sie im Stylesheet notiert haben. Der Längenwert in `margin-top: 2em;` und der Prozentwert in `font-size: 120%;` werden von den verbreiteten grafischen Browsern letztlich in Pixelwerte umgerechnet, sodass `getComputedStyle` Werte mit der Einheit `px`zurückgibt.

Auch beispielsweise bei Farbwerten können Sie nicht erwarten, dass das Format des berechneten Wertes mit dem des Stylesheets übereinstimmt. Denn es gibt in CSS verschiedene Formate, um Farbwerte zu notieren. Notieren Sie im Stylesheet beispielsweise `color: red`, so kann es sein, dass`getComputedStyle` für `color` den Wert `"rgb(255, 0, 0)"` liefert. Dies ist derselbe Wert in einer alternativen Schreibweise.

Microsofts `currentStyle`

`getComputedStyle` wird von allen großen Browsern unterstützt. Der Internet Explorer kennt die Methode jedoch erst ab Version 9. Browserübergreifende Scripte, die ältere Internet Explorer unterstützen, müssen daher eine Sonderlösung für den IE einbauen.

Microsoft bietet eine Alternative, die ähnliches leistet: Jedes Elementobjekt kennt neben dem angesprochenen `style`-Objekt ein gleich aufgebautes Objekt namens `currentStyle` mit Objekteigenschaften für jede unterstützte CSS-Eigenschaft. Im Gegensatz zum `style`-Objekt erlaubt`currentStyle` das Auslesen des aktuellen berechneten CSS-Eigenschaftswertes:

```
var element = document.getElementById('beispielID');
var currentStyle = element.currentStyle;
window.alert("Textfarbe: " + currentStyle.color);
window.alert("Elementbreite: " + currentStyle.width);
```

`currentStyle` liefert meist ein ähnliches Ergebnis wie das standardisierte`getComputedStyle`. In manchen Fällen gibt es jedoch Abweichungen, etwa im obigen Beispiel beim Auslesen des `width`-Wertes. `currentStyle.width` gibt `auto` zurück, wenn dem Element keine explizite Breite zugewiesen wurde. Für das browserübergreifende Auslesen der Box-Größe eignen sich stattdessen

die Eigenschaft offsetWidth/offsetHeight sowie clientWidth/clientHeight.

Browserübergreifendes Auslesen von CSS-Eigenschaften

Durch Kombination von getComputedStyle für standardkonforme Browser und currentStylefür ältere Internet Explorer können wir eine lose Helferfunktion schreiben, die uns den aktuellen CSS-Eigenschaftswert liefert. Die Funktion fragt ab, welches Objekt zur Verfügung steht, und bringt den Wert damit in Erfahrung:

```
function getStyleValue (element, cssProperty) {
    var value = "";
    if (window.getComputedStyle) {
        value = window.getComputedStyle(element, null)[cssProperty];
    } else if (element.currentStyle) {
        value = element.currentStyle[cssProperty];
    }
    return value;
}
```

Bei der Anwendung wird der Funktion das Elementobjekt und ein String übergeben, der den Eigenschaftsnamen in der JavaScript-typischen Schreibweise enthält:

```
var currentFontSize = getStyleValue(document.getElementById("beispielID"), "fontSize");
window.alert(currentFontSize);
```

Dies funktioniert zwar browserübergreifend, allerdings ist das Ergebnis unterschiedlich: Ältere Internet Explorer, in welchen nur currentStyle zur Verfügung steht, geben für die font-size-Eigenschaft einen pt-Wert zurück, andere Browser einen px-Wert. Mit diesen Browserunterschieden müssen Sie rechnen, sie lassen sich nicht einfach vereinheitlichen.

Elementbox-Größen auslesen über Microsoft-Eigenschaften

Über die vorgestellten Techniken ist es nicht browserübergreifend möglich, die aktuelle Höhe und Breite einer Element-Box in der Einheit Pixel auszulesen. Stattdessen sollten Sie folgende Eigenschaften der Elementobjekte verwenden. Sie wurden ursprünglich von Microsoft erfunden, erfreuen sich aber breiter Browser-Unterstützung. Im Gegensatz zu getComputedStyle und currentStyle geben sie keine String-Werte samt Einheiten zurück, sondern direkt JavaScript-Zahlen (Number-Werte) in der Einheit Pixel.

offsetWidth und offsetHeight

liefern die Breite bzw. Höhe der Rahmen-Box des Elements. Das bedeutet, dass der Innenabstand (padding) und der Rahmen (border) inbegriffen sind, der Außenrahmen hingegen (margin) nicht.

clientWidth und clientHeight

liefern Breite bzw. Höhe der Innenabstand-Box des Elements. Das bedeutet, dass paddinginbegriffen ist, während border und margin nicht eingerechnet werden. Ebenso wird die Größe einer möglicherweise angezeigte Bildlaufleiste (Scrollbar) nicht einberechnet.

scrollWidth und scrollHeight

geben die tatsächlich angezeigte Breite bzw. Höhe des Inhalts wieder. Wenn das Element kleiner ist, als der Inhalt es erfordert, also Bildlaufleisten angezeigt werden, so geben diese Eigenschaften die Größe des des aktuell sichtbaren Ausschnittes wieder.

In den meisten Fällen werden Sie die äußere Größe eines Elements benötigen, also offsetWidthund offsetHeight. Das folgende Beispiel gibt die Größe eines Elements aus:

```
var element = document.getElementById('beispielID');
window.alert("Breite: " + element.offsetWidth + "\nHöhe: " + element.offsetHeight);
```

Falls sie die innere Größe benötigen, so können Sie zunächst die aktuellen Werte der jeweiligenpadding-Eigenschaften auslesen. Das sind padding-left und padding-right für die Breite bzw.padding-top und padding-

`bottom` für die Höhe. Diese substrahieren sie von `offsetWidth` bzw.`offsetHeight`, um die tatsächliche Innengröße zu erhalten.

JavaScript: Sicherheit

Einleitung

JavaScript wurde lange als gefährlich und unsicher angesehen, sodass viele Webautoren auf JavaScript verzichteten und viele Websurfer die Ausführung von JavaScripten deaktivierten. Dieser Panikmache sollen hier neutrale Informationen gegenübergestellt werden, ohne zu behaupten, JavaScript sei per se sicher und harmlos.

Die besagten Ängste hatten verschiedene Gründe. JavaScript wird seit seinem Bestehen auch zur Gängelung und Irreführung der Websurfer eingesetzt. Moderne Browser haben deshalb Gegenmaßnahmen ergriffen, die die Möglichkeiten von JavaScripten in verschiedenen Punkten beschneiden. Diese **Einschränkungen** werden wir auf dieser Seite kennenlernen.

Der Einflussbereich der breit akzeptierten Kerntechniken (ECMAScript, das Browser Object Model sowie das Document Objekt Model) ist relativ scharf umrissen. Ein JavaScript, das nur diese Techniken verwendet, hat begrenzte Möglichkeiten und damit ein vergleichsweise geringes Gefahrenpotenzial. Vorausgesetzt ist, dass die Browser **grundlegenden Sicherheitskonzepte** beachten - auch diese werde im Folgenden vorgestellt.

Wenn Sicherheitslücken in Browsern entdeckt werden, ist in den meisten Fällen JavaScript im Spiel. Ein Teil dieser Lücken ermöglicht ein Umgehen der grundlegenden Sicherheitsbeschränkungen, ein anderer betrifft **JavaScript-Erweiterungen**. Denn JavaScript ist mittlerweile ein Türöffner für vielfältige clientseitigen Programmierung, die weit über die besagte Kerntechniken hinausreicht.

Die Browserhersteller sind bemüht, die Fähigkeiten von JavaScript zu erweitern, u.a. indem sie Schnittstellen zu bestehenden Techniken einbauen. Zum Beispiel im Internet Explorer hat JavaScript (JScript) Zugriff auf *ActiveX*-Objekte und den sogenannten *Windows Scripting Host*. Darüber sind - zumindest prinzipiell - sicherheitskritische Zugriffe auf den Client-Rechner möglich. Nun sind diese Schnittstellen nicht für jedes Script verfügbar, sondern durch Sicherungsmechanismen geschützt. Weil diese jedoch in der Vergangenheit zu freizügig waren oder nicht hinreichend funktionierten, entstanden unzählige Sicherheitslücken.

Auch wenn der Internet Explorer solche Probleme mittlerweile im Griff hat: Das Beispiel soll ein allgemeines Problem

verdeutlichen, das fast alle Browser betrifft, und das bisher zu allen Zeiten. JavaScript ist im Hinblick auf Sicherheit nicht unproblematisch und es ist verständlich, wenn Anwender JavaScript deaktivieren oder dessen Möglichkeiten einschränken.

Sicherheitskonzepte von JavaScript

Sandbox-Prinzip

Ein JavaScript verfügt im Vergleich zu anderen Computerprogrammen nur über begrenzte Möglichkeiten. Es operiert im Rahmen eines Browserfenster und eines Dokumentes. Innerhalb dieses strengen Rahmens, in den das Script eingesperrt ist, darf es recht frei schalten und walten, denn es kann nur begrenzten Schaden anrichten. Diese grundlegende Beschränkung nennt sich *Sandbox-* oder**Sandkastenprinzip**.

Insbesondere kann ein gewöhnliches JavaScript auf einer Webseite kann **keine Dateien auf dem Client-Rechner auslesen**, geschweige denn Änderungen daran vornehmen. Es kann auch keine Betriebssystem- oder Browsereinstellungen ändern oder Software auf dem Client-Rechner installieren.

Es gibt nur einige wenige Ausnahmen, in denen ein JavaScript über Browserfenster und Dokument hinaus operieren kann. Zum Beispiel kann es einige bestimmte Browserfunktionen aufrufen und einfache Dialogfenster sowie weitere Browserfenster öffnen. Diese Ausnahmen, die meist mit gewissen Einschränkungen verbunden sind, werden wir noch kennenlernen.

Same-Origin-Policy

Die Same-Origin-Policy (zu deutsch etwa: *Grundregel des selben Ursprungs*) besagt, dass ein JavaScript eines Dokuments nur auf diejenigen anderen, fremden Dokumente zugreifen darf, die dieselbe Herkunft haben. Mit *derselben Herkunft* ist kurz gesagt die Domain in der URI des Dokuments gemeint.

Ein JavaScript hat zunächst einmal Zugriff auf das Dokument, an das es gebunden ist und in dessen Kontext es ausgeführt wird. Bei der Verwendung von Frames, Inner Frames und Popup-Fenstern kann ein Script auch auf andere Dokumente zugreifen. Die Same-Origin-Policy schränkt diese dokumentübergreifenden Zugriffe ein.

Nehmen wir an, in einem Frame wird die URI `http://www.example.org/dokument1.html`geladen und in einem anderen Frame desselben Framesets die URI`http://www.example.org/dokument2.html`. Diese beiden Dokumente haben denselben Ursprungsdomain, nämlich `www.example.org`. Daher können Scripte beider Dokumente gegenseitig auf das jeweils andere Dokument zugreifen, um z.B. Formulardaten oder Cookies auszulesen, über das DOM Änderungen vorzunehmen oder Anwender-Ereignisse zu überwachen.

Wenn die URI des zweiten Dokuments hingegen`http://www.example.net/dokument2.html` lautet, dann sperrt die Same-Origin-Policy den dokumentübergreifenden Zugriff. Denn der Ursprung ist unterschiedlich, einmal `www.example.org`und einmal `www.example.net`.

Ziel der Same-Origin-Policy ist, dass eine Webseite die Daten einer anderen nicht so einfach abgreifen kann. Dies wäre natürlich kein Problem, wenn die andere Webseite sowieso öffentlich ist. Es wäre hingegen ein schwerwiegendes Sicherheitsrisiko bei all denjenigen Webseiten, die einer Anmeldung bedürfen und vertrauliche Daten anzeigen - zum Beispiel Webmail-Dienste, Communities und sämtliche personalisierbaren Webanwendungen.

Die Same-Origin-Policy greift auch bei `XMLHttpRequest`, besser unter dem Namen Ajax bekannt. Mit `XMLHttpRequest` kann ein Script HTTP-Anfragen auslösen und somit Daten vom Webserver empfangen oder an ihn übertragen. Die Same-Origin-Policy sorgt dafür, dass mit `XMLHttpRequest` nur HTTP-Anfragen an dieselbe Domain gesendet werden können.

An einem Punkt greift die Same-Origin-Policy nicht: Ein HTML-Dokument kann mittels `script`-Element JavaScripte von fremden Domains einbinden. Diese werden mit denselben Rechten ausgeführt wie JavaScripte von derselben Domain. Beispielsweise kann`http://www.example.org/dokument1.html` das externe Script mit der URI`http://www.example.net/script.js` einbinden. Diesen Einbinden von Scripten von fremden Webservern Gang und Gäbe vor allem zum Einbinden von Online-Werbung und Statistik-Scripten. Aus der Perspektive der Sicherheit ist eine äußerst zweischneidige Praxis: Einerseits ist es ein sehr nützliches Feature, denn es macht z.B. die Nutzung von Webdiensten möglich. Andererseits kann es zu schwerwiegenden Sicherheitslücken führen, fremden Code in die eigene Seite einzubinden – wir werden später beim Cross-Site-Scripting darauf zurückkommen.

Same-Origin-Policy und Subdomains

Die Same-Origin-Policy blockt nicht nur den Zugriff, der sogenannte Second-Level-Domains übergreift (z.B. `example.org` darf nicht auf `example.net` zugreifen). Die Sperre blockt auch den Zugriff zwischen Subdomains derselben Domains. Das heißt, ein Script in einem Dokument unter`de.example.org` hat keinen Zugriff auf ein Dokument unter `en.example.org`, obwohl die Domain dieselbe ist (`example.org`) und sich bloß die Subdomain unterscheidet (*de* gegenüber *en*).

Diese Regelung mag zunächst rigide und streng scheinen, ist aber eine wichtige Sicherheitsbarriere. Diese Sperre geht davon aus, dass unter einer Domain verschiede Websites liegen können, die ihre Daten nicht miteinander teilen wollen. Selbst wenn beide Domains zu einer Site gehören, lassen sich die verschiedenen Domains auf diese Weise kapseln und absichern.

Es gibt jedoch die Möglichkeit, dass ein Dokument einwilligt, dass es für den Zugriff von derselben Domain offen ist.

In einem Dokument unter `de.example.org` wird folgende JavaScript-Anweisung notiert:

```
document.domain = "example.org";
```

Damit ist das Dokument für Scripte zugänglich, die auf einer Domain liegen, die auf`example.org` endet. Also nicht nur für `de.example.org`, sondern auch für `en.example.org` oder`hildegard.de.example.org`.

Dieses Schema gilt nicht nur für Second-Level-Domains, sondern für beliebige Subdomains. Ein Script unter `hildegard.de.example.org` kann folgende Anweisung notieren:

```
document.domain = "de.example.org";
```

Damit erlaubt es den Zugriff z.B. von `mechthild.de.example.org` und allen anderen Domains, die auf `de.example.org` enden.

Local Machine Zone Lockdown (Internet Explorer)

Die Same-Origin-Policy lässt einen Punkt außer Acht: Ein Script darf im Kontext der Herkunftsdomain ohne Begrenzung schalten und walten sowie mittels `XMLHttpRequest` Daten empfangen und versenden. Das kann zu einem schwerwiegenden Problem werden, wenn das Script nicht im Web, sondern *lokal* ausgeführt wird. *Lokal* bedeutet, dass das Dokument auf einer Festplatte des Client-Rechners liegt und von dort aus im Browser geöffnet wird. Die URI beginnt dann mit`file://localhost/`, in der Kurzschreibweise `file:///`.

Die Konsequenz ist, dass ein solches Script prinzipiell **alle** Dateien auf den erreichbaren Datenträgern auslesen kann (aber nicht ändern - zumindest nicht über `XMLHttpRequest` alleine). Mit einigen Kniffen können diese abgegriffenen Daten ins Web gesendet werden. Somit ließen sich vertrauliche Daten ausspionieren. Es stellt daher ein grundlegendes Problem dar, wenn fremde Dokumente mit JavaScript auf den eigenen Rechner gelangen.

Der Internet Explorer ab Windows XP mit dem Service Pack 2 stellt daher alle lokalen Dokumente mit Scripten unter Generalverdacht und verhindert ihre Ausführung. Dieser Sicherheitsmechanismus nennt sich *Local Machine Zone Lockdown*, zu deutsch *Sperrung der Zone des lokalen Computers*.

Wie sich dieser Generalverdacht auswirkt und wie man den Internet Explorer trotzdem dazu bringen kann, Scripte in lokalen Dokumenten auszuführen, erörtert der Artikel Umgehung der Sperrung der lokalen Zone.

Browser-Einschränkungen und Schutz vor schädlichen JavaScripten

JavaScript hat zwar keine vollständige Kontrolle über den Client-Rechner und den Browser, besitzt aber einige Möglichkeiten des Missbrauchs, mit denen der Benutzers irregeführt, belästigt und gegängelt werden kann. Mittlerweile besitzen die Browser eingebaute Schutzmechanismen, die gewisse Freiheiten von JavaScripten beschränken. Sie sollten diese kennen, denn sie werden bei der JavaScript-Entwicklung früher oder später an diese Grenzen stoßen.

Popup-Blocker

Ein problematisches Thema ist das Öffnen von neuen Fenster mit `window.open`. Diese Methode wird unter anderem dazu missbraucht, um sogenannte **Popup-Fenster** (kurz: *Popups*) mit Werbung zu öffnen, die automatisch und ohne ausdrücklichen Wunsch des Websurfers aufspringen. Das unkontrollierte Öffnen von Fenstern belästigt den Surfer nicht nur, sondern ist auch ein

Sicherheitsproblem, denn es kann den Browser lahmlegen oder sogar zum Abstürzen bringen.

Aus diesem Grund haben mittlerweile alle Browser einen sogenannten **Popup-Blocker** eingebaut. Ältere Browser lassen sich mit entsprechenden Zusätzen nachrüsten. Diese Blocker erlauben das Öffnen von Fenstern mittels JavaScript nur, wenn damit *auf eine Benutzereingabe reagiert wird*. Wenn sie also einfach `window.open` aufrufen, werden die meisten Popup-Blocker das Öffnen des Fensters unterbinden:

```
<script type="text/javascript">
window.open("dokument.html", "fenstername");
</script>
```

Wenn Sie ein Fenster jedoch im Zuge der JavaScript-Behandlung (*Event-Handling*) einer Benutzereingabe öffnen, erlauben es die Popup-Blocker üblicherweise. So können Sie beispielsweise ein`a`-Element mit einem `click`-Handler versehen. Ein einfaches Beispiel mit eingebetteten Event-Handler-Attributen sähe so aus:

```
<a href="dokument.html" onclick="window.open(this.href, 'popup')">
   Dokument XYZ im eigenen Fenster öffnen
</a>

<button type="button" onclick="window.open('dokument.html', 'popup')">
   Dokument XYZ im eigenen Fenster öffnen
   </button>
```

Sie können den `click`-Handler alternativ gemäß dem Traditionellen Event-Handling registrieren:

```
function popupFenster (adresse) {
   window.open(this.href, 'popup');
}

window.onload = function () {
   document.getElementById("popupLink").onclick = popupFenster;
};
```

Vorausgesetzt bei diesem Beispiel ist, dass im HTML-Code ein Element mit `id="popupLink"`existiert.

Popup-Blocker versuchen zwischen *erwünschten* und *unerwünschten* Popup-Fenstern zu unterscheiden. Ein Browser kann nicht zuverlässig unterscheiden, ob ein Fenster vom Anwender erwünscht ist oder nicht. Das angesprochene Kriterium der *Benutzereingabe* (z.B. ein Mausklick auf ein Element) ist nur bedingt zur Unterscheidung tauglich: Manche Webseiten gaukeln dem Browser vor, sie würden ein »erwünschtes« Popup-Fenster als Reaktion auf eine Benutzereingabe öffnen, indem sie z.B. beim Klick irgendwo ins Dokument zusätzlich ein Werbe-Popup öffnen.

Es gibt keine allgemeingültigen Regeln, nach denen die verschiedenen Popup-Blocker arbeiten. Zudem können sie verschieden »scharf« eingestellt werden. Es ist daher schwierig, zuverlässige Aussagen darüber zu treffen, welche Popup-Fenster geblockt und welche zugelassen werden.

Trotzdem ein paar grobe **Empfehlungen**: Sie sollten darauf verzichten, Fenster als Reaktion auf die dokumentweite Ereignisse zu öffnen. Das betrifft die Ereignisse `load` oder `unload`, aber auch Mausereignisse wie `click` oder Tastaturereignisse wie `keypress` bei zentralen Objekten wie `window`und `document` sowie bei den HTML-Elementen `html` und `body`. Solche Fenster werden höchstwahrscheinlich geblockt. Wenn Sie punktuell Popup-Fenster öffnen wollen, dann geben sie einem`a`- oder `button`-Element einen Event-Handler für das `click`-Ereignis. Das obige Beispiel illustriert dies.

Die Veränderung der Fenstereigenschaften

Das Öffnen von neuen Fenstern bringt noch weiteres Missbrauchspotenzial und **schwerwiegende Sicherheitsprobleme** mit sich. Ursprünglich war es möglich, dass ein Script volle Kontrolle über das Aussehen und das Verhalten des neuen Fensters hatte. Die `window.open`-Methode hat für diese Fensteroptionen einen dritten Parameter. Problematische `window.open`-Aufrufe sehen zum Beispel so aus:

```
window.open("dokument.html", "popup1", "top=1000,left=1000,width=10,height=10")

window.open("dokument.html", "popup2", "location=no,menubar=no,resizable=no,status=no,toolbar=no")
```

`window.open` hatte direkten Einfluss auf die **Größe des Fensters**, dessen **Position auf dem Bildschirm**, aber auch auf die **Anzeige der browsertypischen Bedienelemente**. Auch war es möglich, Fenster ohne Einschränkungen nachträglich in ihrer **Größe und Position** zu verändern, sodass man sie beliebig über den Bildschirm verschieben konnte (mittels `window.resizeBy`, `window.resizeTo`sowie `window.innerHeight` und `window.innerWidth`). Gleichzeitig ließ sich unterbinden, dass der Anwender das Fenster in der Größe verändern konnte.

Sie können sich den Missbrauch vorstellen, der dadurch ermöglicht wurde: Indem eine winzige oder überdimensionierte Größe und eine Position außerhalb des Bildschirmes angegeben wurde, konnte der Anwender das Fenster nicht sehen geschweige denn es auf die gewohnte Art schließen. Oder das Fenster hüpfte immer weg, sobald es der Anwender schließen wollte.

Das Verstecken der Menü-, Symbol-, Adress- und Statusleisten wurde auf breiter Front missbraucht, um Websurfer vorzugaukeln, er befinde sich auf der Login-Seite einer anderen, ihm bekannten und vertraulichen Webseite. Auf diese Weise werden im großen Stil persönliche Daten gestohlen - im Fachjargon nennt man diesen Datenklau Phishing.

Eine besonders perfide Gänglung des Benutzers erlaubten alte Versionen des Internet Explorers: Mit der Angabe der Fensteroption `fullscreen=yes` konnte ein Popup-Fenster im Vollbildmodus geöffnet werden. Über einen solchen *Kiosk-* oder *Präsentationsmodus* verfügen auch andere Browser, allerdings war es JavaScripten in anderen Browsern nicht erlaubt, diesen selbstständig zu aktivieren. Im Vollbildmodus war auf dem Bildschirm nichts als die Webseite zu sehen, alles andere wurde überlagert.

Neuere Browser schränken aus diesen Gründen die Einflussmöglichkeiten von nicht priviligierten JavaScripten auf die Darstellung von Browserfenstern stark ein. Gewisse Leisten können per JavaScript nicht mehr ausblendet werden oder sind zumindest immer in einer Kompaktdarstellung zu sehen. Insbesondere die **Adressleiste** wird immer angezeigt, sodass der Anwender stets weiß, auf welcher Webseite er sich befindet, und entscheiden kann, ob sie vertrauenswürdig ist. Viele Browser sorgen außerdem dafür, dass das Fenster eine **Mindest- und Maximalgröße** hat, auf dem Bildschirm tatsächlich zu sehen ist und der Anwender dessen **Größe frei verändern** kann.

Aus Gründen der Benutzerfreundlichkeit sei Ihnen ohnehin geraten, die Browser-Bedienelemente nicht zu verbergen. Je nachdem, was Sie im Popup-Fenster anzeigen möchten, ist der Benutzer dankbar, wenn er über die vertrauten Navigationsmöglichkeiten verfügt. Verzichten Sie möglichst darauf, die Browserleisten im dritten Parameter von `window.open` auszuschalten. Neuere Browser ignorieren viele dieser Angaben ohnehin und bestimmen die Anzeige von Menü und Leisten selbst. Das genaue Resultat können Sie nicht zuverlässig abschätzen, denn diese JavaScript-Einschränkungen unterscheiden sich von Browser zu Browser und sind individuell konfigurierbar.

Die beschriebenen Probleme mit Popup-Fenstern und die Gegenmaßnahmen seitens der Browser haben dazu geführt, dass der Einsatz von Popup-Fenstern nach und nach zurückgegangen ist. Es gibt noch weitere Gründe, warum Popup-Fenster aus der Mode sind. Einer davon ist, dass moderne Browser ihr Fensterkonzept komplett umgemodelt haben. Früher wurde in einem eigenständigen Browserfenster genau ein HTML-Dokument genau dargestellt. Heutzutage bieten die meisten grafischen Browser**Tabbed Browsing**. Das heißt, sie stellen mehrere Dokumente innerhalb eines Fensters dar und machen diese über Registerkarten zugänglich.

Die problematischen Fensterveränderungen, die wir betrachtet haben, verlieren beim *Tabbed Browsing* ihren Sinn. Da klassische Popup-Fenster das Konzept von Registerkarten durchbrechen, überlassen Browser zunehmend dem Anwender die Wahl, ob `window.open` ein eigenständiges Fenster oder eine Registerkarte öffnet. Auf deren Darstellung hat der Autor des JavaScriptes immer weniger Einfluss - zu Gunsten des Anwenders.

Kontextmenü und rechte Maustaste

Als Kontextmenü wird das Aufklappmenü bezeichnet, das üblicherweise dann erscheint, wenn der Anwender ein Element des HTML-Dokuments mit der rechten Maustaste anklickt. Je nach Hardware, Betriebssystem und Browser gibt es noch weitere Möglichkeiten, das Kontextmenü aufzurufen.

Dieses Kontextmenü ist für den Anwender enorm praktisch bei der Bedienung einer Webseite. Im Kontextmenü eines Links kann er zum Beispiel wählen, dass das Linkziel in einem neuen Fenster geöffnet wird oder die Zieladresse in die Zwischenablage kopiert wird.

Dessen ungeachtet versuchen zahlreichen Webseiten, mittels JavaScript die Anzeige dieses Kontextmenüs im gesamten Dokument zu unterbinden. Diese Scripte reagieren dokumentweit auf die Ereignisse `contextmenu` und `onmousedown` und

unterdrücken die Standardaktion des Browsers. Die Autoren wollen damit verhindern, dass Texte oder Bilder kopiert werden können oder der HTML-Quellcode gelesen werden kann. Meist wollen sie sich damit gegen eine urheberrechtswidrige Weiterverwendung der eigenen Werke schützen.

Es kann nüchtern festgestellt werden, dass das Sperren des Kontextmenüs diesen Zweck nicht zuverlässig erfüllt. Stattdessen richtet es mehr Schaden als Nutzen an. Wer Texte und Bilder kopieren möchte bzw. den Quelltext lesen will, schafft es ohne viel technisches Know-How auch trotz dieser »Rechtsklick-Sperre«.

Neuere Browser haben erkannt, dass das Sperren des Kontextmenüs den Benutzer gängelt und in der gewohnten Bedienung von Webseiten einschränkt. Sie bieten daher in ihrer Konfiguration die Möglichkeit, diese Sperren zu ignorieren. »Rechtsklick-Sperren« werden damit schlichtweg wirkungslos.

Es mag in besonderen Fällen, insbesondere speziellen Webanwendungen, seinen Sinn haben, ein eigenes, angepasstes Kontextmenü bereitzustellen. Aus diesem Grund ermöglichen verschiedene Browser die Behandlung des `contextmenu`-Ereignisses. Aber auch in dem Fall ist das Unterdrücken des browsereigenen Kontextmenüs nur möglich, wenn eine entsprechende Browsereinstellung es zulässt.

...

Lang laufende Scripte

..

Browser-Einschränkungen konfigurieren

... an welchen Stellen man das JavaScript-Verhalten der Browser einstellen kann.

IE 8: Extras > Popup-Blocker > Popupblockereinstellungen; Internetoptionen > Erweitert; Internetoptionen > Sicherheit > [Zone] > Skripting / Verschiedenes

Firefox 3.0: Extras > Einstellungen > Inhalt > JavaScript aktivieren > Erweitert...

Opera: Tools > Preferences > Advanced > Content > JavaScript Options...

Zonenmodelle, Positivlisten und seitenspezifische Einstellungen

Ein wichtiges Sicherheitsfeature von Browsern sind Website-spezifische JavaScript-Einstellungen. Je nachdem, welche Website angesurft wird, wird die Ausführung von JavaScripten uneingeschränkt zugelassen, nur eingeschränkt zugelassen oder der JavaScript-Interpreter wird komplett deaktiviert und Scripte gar nicht ausgeführt. Dies trägt dem Umstand Rechnung, dass JavaScript als Haupteinfallstor für die Ausnutzung von Browser-Sicherheitslücken dient, zur Gängelung des Anwenders missbraucht wird oder enfach unerwünschte Werbung einbindet.

Diese seitenspezifischen Einstellungen sind von Browser zu Browser unterschiedlich umgesetzt und betreffen nicht nur JavaScript, sondern auch andere sicherheits- und datenschutzkritische Techniken wie Cookies und Plugins.

Internet Explorer

Der Internet Explorer verfügt über verschiedene Sicherheitszonen, die standardmäßig an gewisse Einstellungen gekoppelt sind. Eine normales HTML-Dokument im World Wide Web liegt in der*Internetzone*, ein Dokument auf dem lokalen Rechner oder im lokalen Netzwerk in der Zone *Lokales Intranet*.

Frage von: mschaefer

Das kann so nicht stimmen, Local Machine Zone Lockdown nicht berücksichtigt?! Wie spielt die darein?

Daneben existieren zwei Zonen, zu denen der Anwender eigenständig Webadressen und Netzwerk-Pfade hinzufügen kann: *Vertrauenswürdige Sites* und *Eingeschränkte Sites*. Dies erlaubt dem Anwender beispielsweise, für die Internetzone eher restriktive Sicherheitseinstellungen zu wählen, die dann für bestimmte Seiten gelockert werden können.

ToDo von: mschaefer

Screenshots

Firefox

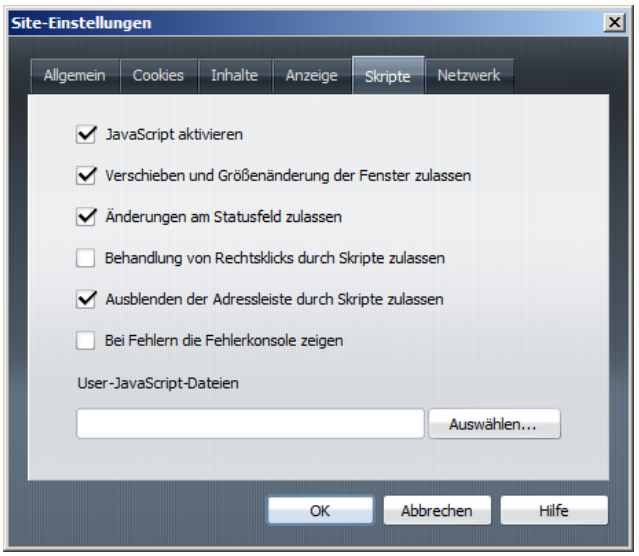
Mozilla Firefox verfügt intern über seitenspezifische Einstellungen, bietet standardmäßig aber keine Menü an, über das der Anwender die Einstellungen komfortabel regulieren könnte. Der Firefox-Zusatz NoScript erfreut sich jedoch einiger Verbreitung. Dieser erlaubt das seitenweise Erlauben oder Verbieten der Ausführung von JavaScripten und kann Scripten weitere Beschränkungen auferlegen.

ToDo von: mschaefer

Screenshot NoScript

Opera

Im Opera können Sie eine Vielzahl von Einstellung seitenspezifisch anpassen. Navigieren Sie zunächst zur Webseite, für die Sie besondere Einstellungen angeben wollen. Klicken Sie mit der rechten Maustaste auf eine freie Fläche des Dokuments und wählen Sie im Kontextmenü den Eintrag*Seitenspezifische Einstellungen...*. Unter der Registerkarte *Skripte* können Sie nicht nur JavaScript für die Seite aktivieren oder deaktivieren, sondern auch verschiedene JavaScript-Einstellungen festlegen:



Safari

...

Privilegien und Signaturen (Gecko)

...

Cross-Site Scripting (XSS)

Cross-Site Scripting, abgekürzt XSS, ist das Einschleusen von fremden, möglicherweise schädlichen JavaScripten in eine Website. Es handelt sich weniger um ein Sicherheitsproblem innerhalb von JavaScript, sondern um eine Sicherheitslücke in fehlerhaften Webanwendungen. Wenn Webanwendungen Daten aus nicht vertrauenswürdigen Quellen (z.B. aus Formulareingaben oder HTTP-Parametern) ungefiltert ins HTML einbauen, so können Angreifer schlimmstenfalls dauerhaft (persistent) JavaScript-Code einschmuggeln.

Dieser Code wird mit allen Rechten ausgeführt, die ein JavaScript üblicherweise hat. Handelt es sich um eine per Login geschützte Anwendung, so kann das Scriptdie Benutzer-Beglaubigung missbrauchen und im Namen des Benutzers automatisiert Aktionen vornehmen. Denn das eingeschleuste Script kann HTTP-Anfragen an die Domain versenden, darüber private Daten auslesen, ändern und verschicken. Weder muss der Benutzer davon Notiz nehmen, noch kann die Webanwendung ohne weiteres unterscheiden,

ob ein schädliches JavaScript sogenanntes *Session-Hijacking* betreibt oder der Benutzer selbst Urheber der Aktionen ist.

XSS-Lücken in großen Webanwendungen wie MySpace, Facebook, Orkut, StudiVZ und Twitter haben spektakuläre JavaScript-Würmer möglich gemacht. Diese pflanzten sich innerhalb der Website z.B. über Benutzerprofile fort, konnten private Daten auslesen oder löschen (Phishing) und damit großen Schaden anrichten. Es gibt auch XSS-Würmer, die andere Domains mit derselben Webanwendung (z.B. der Blogsoftware WordPress) infizierten und sich so über Webserver hinweg verbreiteten.

Um XSS-Lücken zu vermeiden, ist eine sorgfältige Prüfung, Filterung und Entschärfung aller nicht vertrauenswürdiger Daten nötig, die in den HTML-, CSS- und JavaScript-Code server- oder clientseitig eingebaut werden.

JavaScript: Serverkommunikation und dynamische Webanwendungen (Ajax)

Entstehung von Ajax: Neue dynamische Webanwendungen

In der Anfangszeit bestand das World Wide Web aus einer Sammlung von weltweit abrufbaren wissenschaftlichen Daten und Dokumenten. Das heutige Web hingegen bietet nicht bloß statische Informationsangebote, sondern maßgeblich interaktive Dienste. Spätestens unter dem Schlagwort *Web 2.0* sind unzählige sogenannte *Webanwendungen* aus dem Boden gesprossen, die vorher ungeahnte Dienste ins Web brachten. Man denke nur an Kontaktnetzwerke, die selbst keine Inhalte bereitstellen, sondern ihren Nutzern Kommunikation und das Einstellen eigener Inhalte ermöglichen. Das Web als Verbund von mehreren Rechnern, nicht mehr der einzelne Rechner ist die Plattform für Software dieser Art.

Viele dieser neuen Webanwendungen setzen stark auf JavaScript, um die Funktionalität klassischer Desktop-Anwendungen bereitzustellen. Heraus kommen zum Beispiel E-Mail-Programme, News-Leseprogramme, Textverarbeitung, Tabellenkalkulation, Präsentationen, Chat-Programme, Foto-Verwaltung und Bildbearbeitung, Terminkalender und Adressbücher, Karten-Anwendungen und vieles mehr. Dadurch, dass die Anwendung nun im Netz beheimatet ist, ergeben sich besondere Mehrwerte: Man spricht von »sozialer« und »kollaborativer« Software, deren Fokus auf Kommunikation, gemeinsame Nutzung von Daten und Zusammenarbeit liegt. Mittlerweile genießen diese Anwendungen eine ungeheure Popularität und vereinfachen die Arbeit am Computer und im Netz.

JavaScript spielt dabei eine zentrale und neue Rolle. Es handelt sich nicht um klassische HTML-Dokumente, denen mit JavaScript ein wenig Interaktivität hinzugefügt wird. Stattdessen funktionieren viele dieser Webanwendungen nicht ohne JavaScript. Durch komplexes Event-Handling und viele Tricks bringt JavaScript einfache HTML-Elemente dazu, sich wie Bedienelemente von Desktop-Programmen zu verhalten - z.B. wie Schaltflächen, Menüs oder Dialogfenster.

Was ist anders an Ajax?

Gemeinsam ist den vielen der besagten Webanwendungen eine Schlüsseltechnik namens **Ajax**. Das bedeutet: **JavaScript tauscht im Hintergrund Daten mit dem Webserver aus**. Das funktioniert über selbst erzeugte HTTP-Anfragen, deren Server-Antwort dem Script zur Verfügung steht.

Was ist daran nun neu? Dazu muss man zuerst verstehen, wie Interaktion im Web *ohne Ajax*funktioniert: Herkömmliche Websites nutzen Links und Formulare, um mit dem Webserver zu interagieren. Der Anwender aktiviert einen Link oder sendet ein Formular ab, woraufhin der Browser eine entsprechende HTTP-Anfrage an den Webserver sendet. Der Webserver antwortet, indem er üblicherweise ein HTML-Dokument zurückliefert, das der Browser verarbeitet und anstelle des alten anzeigt. Ohne Ajax muss also immer ein neues, vollständiges HTML-Dokument vom Server geladen werden. Diese HTML-Dokumente werden oft in Webanwendungen oftmals von serverseitigen Programmen generiert.

Ajax durchbricht dieses Prinzip und kann damit die Bedienung von Webseiten und den Aufbau von Webanwendungen grundlegend ändern. Es werden nicht immer neue HTML-Dokumente heruntergeladen und ausgewechselt, sondern nur kleine Datenportionen mit dem Webserver ausgetauscht. Gerade benötigte Daten werden nachgeladen und ausgewählte Änderungen dem Server mitgeteilt.

Im Extremfall kommt eine sogenannte *Single Page Application* heraus, bei der es nur *ein*ursprüngliches HTML-Dokument gibt und der restliche Datenaustausch mit dem Webserver per JavaScript im Hintergrund abläuft. Über die DOM-Schnittstelle wird das Dokument nach Belieben umgestaltet. Es reagiert auf Benutzereingaben, übersendet diese gegebenenfalls an den Server, lädt im Hintergrund Inhalte vom Server nach und montiert diese ins bestehende Dokument ein.

Der Begriff »Ajax« und seine Schwächen

Der Begriff *Ajax* wurde ursprünglich im Jahr 2005 von dem richtungsweisenden Artikel A New Approach to Web Applications von Jesse James Garrett geprägt. Ajax steht darin als Abkürzung für*Asynchronous JavaScript and XML* (auf Deutsch: asynchrones JavaScript und XML).

Diese Abkürzung stift leider mehr Verwirrung, als sie zum Verständnis beiträgt. Weder sind Ajax-Anwendungen asynchron in dem Sinne, dass die Kommunikation mit dem Server völlig losgelöst von Benutzereingaben stattfindet. Noch ist XML zwangsläufig das Übertragungsformat für Daten zwischen Client und Server. Garretts Konzept taugt wenig zum Verständnis der gegenwärtigen Praxis, die unter dem Schlagwort Ajax zusammengefasst wird.

In den meisten Fällen bezeichnet »Ajax« lediglich den JavaScript-gestützten Datenaustausch mit dem Webserver. XML in dabei nur ein mögliches, aber nicht das zentrale Übertragungsformat. Und asynchron bedeutet lediglich, dass die JavaScript-Ausführung beim Warten auf die Server-Antwort nicht den Browser blockiert, sondern dass JavaScript-Ereignisse gefeuert werden, wenn die Server-Antwort eingetroffen ist.

Vor- und Nachteile von Ajax

Klassisches Modell mit eigenständigen, adressierbaren Dokumenten

Das **herkömmliche Modell** funktioniert nach dem Grundsatz »Stop and Go«: Der Anwender klickt Anwender auf einen Link oder den Absende-Button eines Formulars und muss erst einmal warten. Der Browser übermittelt derweil eine Anfrage an den Webserver. Dieser speichert gegebenenfalls Änderungen ab und generiert ein neues, vollständiges Dokument. Erst wenn dieses zum Client-

Rechner übertragen wurde und der Browser es vollständig dargestellt hat, kann der Anwender in der Webanwendung weiterarbeiten.

Der Browser zeigt also beim klassischen Modell eine Reihe von *eigenständigen HTML-Dokumenten* (Ressourcen) an, die alle eine *eindeutige, gleichbleibende Adresse* (URI) besitzen. Dafür stellt der Browser Navigationsmechanismen wie den Verlauf (auch History genannt) zur Verfügung. Der bestechende Vorteil dieses Modells: Diese Dokumente sind unabhängig von JavaScript lesbar, verlinkbar, durch Suchmaschinen indizierbar, problemlos abspeicherbar und so weiter.

Besonderheiten bei Ajax

Mit **Ajax** hingegen werden die Server-Anfragen im Hintergrund gestartet, ohne dass das Dokument ausgewechselt wird. Damit fällt das Warten auf die Server-Antwort entweder ganz weg, weil nicht auf sie gewartet werden muss, oder der Server muss nur eine kleine Datenportion zurückschicken. Der Vorteil von Ajax-Webanwendungen ist daher, dass sie schneller auf Benutzereingaben reagieren und dem vertrauten Verhalten von Desktop-Anwendungen näherkommen.

Ajax bricht absichtlich mit grundlegenden Funktionsweisen und Regeln des Webs. Daraus zieht es seine Vorteile, aber auch schwerwiegende Nachteile: Es gibt keine vollständigen, adressierbaren Dokumente mehr, die in Webanwendungen einen bestimmten Punkt und Status markieren. Wenn der Anwender zu dem Inhalt kommen möchte, den er zuvor gesehen hat, betätigt er aus Gewohnheit die Zurück-Funktion. Das funktioniert bei der Verwendung von Ajax nicht mehr wie gewohnt: Denn im Browser-Verlauf wird die Änderung der Inhalte via JavaScript nicht registriert, denn das Dokument ist nicht ausgewechselt worden und die Adresse hat sich nicht geändert. Eine Navigation, wie sie der Anwender von statischen Webseiten gewohnt ist, ist auf Seiten mit solcher JavaScript-Interaktivität nicht ohne weiteres möglich.

Zugänglichkeit von Ajax-Anwendungen

Ajax-Anwendungen verlagern einen großen Teil der Datenverarbeitung vom Server-Rechner auf den Client-Rechner, genauer gesagt in den Browser. Damit steigen die Ansprüche, die an die Zugangssoftware gestellt werden - auch wenn mittlerweile alle neueren JavaScript-fähigen Browser über leistungsfähige Ajax-Umsetzungen verfügen.

Es ist eine besondere Herausforderung, eine Site mit Ajax zugänglich für Nutzer ohne JavaScript, mit alternativen oder assistiven Zugangstechniken wie Screenreadern oder Mobilbrowsern zu gestalten. Funktionen wie der Verlauf, den bisher der Browser automatisch zur Verfügung stellte, müssen in Ajax-Anwendungen nachgebaut werden, z.B. indem jeder Status eine Adresse bekommt, damit die Zurück-Navigation funktioniert und der Status verlinkt werden kann. Es ist also mit einigem Aufwand verbunden, eine Ajax-Anwendung so komfortabel und robust zu bekommen, wie es klassische Lösungen von Haus aus sind.

...

Typische abwärtskompatible Anwendungsfälle von Ajax

Neben vollständigen Webanwendungen, die das Verhalten einer Desktop-Anwendung komplett in JavaScript zu emulieren versuchen, gibt es viele kleine Fälle, in denen Hintergrund-Serverkommunikation auf klassischen Webseiten sinnvoll ist und die Bedienung vereinfacht. In diesen Fällen kann Ajax zumeist als Zusatz verwendet werden. Das heißt: Falls JavaScript verfügbar ist, genießt der Anwender einen gewissen Extra-Komfort in der Bedienung. Falls JavaScript nicht aktiv oder Ajax nicht verfügbar ist, dann kann die Website trotzdem ohne funktionale Einschränkungen benutzt werden. Dieser abwärtskompatible Einsatz entspricht dem Konzept des Unobtrusive JavaScript.

Laden von kleinen Inhaltsfragmenten

Oftmals werden Inhalte in kurzer, übersichtlicher Listenform dargestellt. Um den vollen Eintrag zu sehen, muss ohne Ajax ein neues Dokument geladen werden. Mit Ajax können Listenansicht und Vorschau- bzw. Vollansicht kombiniert werden, ohne dass alle Detail-Informationen von Anfang an versteckt im Dokument liegen. Auf Benutzerwunsch (z.B. beim Klicken oder beim Überfahren mit der Maus) können die Informationen zu einem Eintrag via Ajax nachgeladen werden und erscheinen dann direkt beim entsprechenden Listeneintrag.

Automatische Vervollständigung (Autocomplete und Suche bei der Eingabe)

Ohne Ajax sind Suchmasken mitunter langsam und zäh bedienbar: Man gibt einen Suchbegriff ein, sendet das Formular ab und wartet auf die Trefferliste. In vielen Fällen muss man die Suche verfeinern oder den Suchbegriff korrigieren, z.B. weil man ihn offensichtlich nicht korrekt geschrieben hat. Schneller zum Ziel kommt man, wenn schon beim Tippen die Eingabe im Hintergrund an den Server gesendet wird und unter dem Eingabefeld blitzschnell Suchvorschläge angezeigt werden, die zu der bisherigen Eingabe

passen.

Stellen Sie sich etwa eine Fahrplanauskunft vor: Wenn Sie »Paris« in das Zielfeld eingeben, werden sofort alle Pariser Bahnhöfe aufgelistet, sodass sie einen davon wählen können. Oder Sie geben in einem Produktkatalog einen Herstellernamen, eine Produktkennziffer oder ein Merkmal ein, so kann Ajax mithilfe einer intelligenten serverseitigen Suchfunktion die passenden Produkte sofort anzeigen, noch während sie tippen.

Server-Aktionen ohne Antwort-Daten

Nicht jede Benutzeraktion auf einer Site startet das Abrufen von neuen Informationen vom Server. Es gibt viele Aktionen, die dem Server bloß eine Statusänderung mitteilen, ohne dass dazu das aktuelle Dokument ausgetauscht und ein neues aufgebaut werden muss. All diese sind Kandidaten für sinnvollen Ajax-Gebrauch:

- Einfache Formulare, die per Ajax automatisch abgesendet werden, z.B. bei einer Kommentarfunktion
- Bewertungen mit einer einfachen Skala (z.B. 1-5 Sterne)
- Abstimmungen
- Listen-Funktionen wie Markieren, Ausblenden, Löschen usw.

Beim erfolgreichen Übermitteln der Aktion an den Server gibt dieser üblicherweise nur eine Bestätigung zurück, das dem Benutzer nicht einmal präsentiert werden muss. Daher kann man dem Benutzer standardmäßig eine Erfolgsmeldung zeigen, ohne dass auf die Server-Antwort gewartet werden muss. Nur wenn diese negativ ausfällt, wird eine Fehlermeldung angezeigt. Dadurch wirkt ein Ajax-Interface besonders schnell bedienbar und reagiert ohne Verzögerung auf Benutzereingaben – vorausgesetzt, dass die HTTP-Anfrage korrekt übermittelt wurde und das serverseitige Programm sie verarbeiten konnte.

Blättern und Seitennavigation

Beim Navigieren durch Listen z.B. mit Suchresultaten, Artikeln oder Produkten gibt es üblicherweise eine Seitennavigation. Seite 1 zeigt etwa die Resultate 1 bis 10, Seite 2 zeigt 11 bis 20 und so weiter. Ajax kann das Durchstöbern dieser Listen vereinfachen, indem beim Blättern nicht notwendig ein neues Dokument vom Server geladen werden muss. Stattdessen kann Ajax die Einträge der folgenden Seite schrittweise hinzuladen und z.B. ans Ende der bestehenden Liste einfügen. Das kann sogar soweit gehen, dass die folgenden Einträge automatisch nachgeladen werden, sobald der Anwender an das Ende der gerade angezeigten Einträge scrollt.

Eingebettete Formulare

Vom Benutzer veränderbare Daten (z.B. ein Kundenprofil oder Einstellungen) werden üblicherweise in zwei Ansichten angezeigt: Die Nur-Lesen-Ansicht einerseits und die Editieren-Ansicht andererseits. Beispielsweise gibt es eine tabellarische Auflistung sowie ein zusätzliches Formular, in dem dieselben Daten verändert werden können.

Mit Ajax können beide Ansichten zu einer zusammengefasst werden (sogenannte *Edit-in-place-Formulare* bzw. *Inline Edit*). Will der Benutzer ein Datenfeld editieren, so kann JavaScript die Lese-Ansicht auf Knpofdruck dynamisch in eine Formular-Ansicht wechseln. Hat der Benutzer das Editieren beendet, so werden die Änderungen per Ajax an den Server gesendet. Verlässt der Benutzer die Seite, so bleiben die Änderungen gespeichert und es besteht keine Gefahr, dass der Benutzer vergisst, das Formular abzusenden.

Regelmäßiges Aktualisieren vom Server (Liveticker, E-Mail, Chats)

Anwendungen wie Liveticker oder webbasierte E-Mail-Leseprogramme basieren darauf, dass eine Webseite häufig Aktualisierungen erfährt. Anstatt immer das gesamte Dokument neu zu laden, um eventuelle neue Inhalte anzuzeigen, kann Ajax regelmäßig im Hintergrund beim Server nachfragen, ob seit der letzten Aktualisierung neue Inhalte hinzugekommen sind. Falls ja, sendet der Server diese neuen oder geänderten Einträge in der Antwort gleich mit und JavaScript stellt sie im aktuellen Dokument dar. Ohne dass der Benutzer etwas tun muss, aktualisiert sich die Webseite von selbst und zeigt neue Nachrichten schnellstmöglich an.

Mit Ajax ist bisher keine echte Echtzeit-Aktualisierung möglich, wie sie für Web-basierte Chats und Instant Messaging benötigt wird. Das liegt hauptsächlich daran, dass das Protokoll HTTP auf einem Anfrage-Antwort-Schema anstatt auf einer dauerhaften Verbindung zwischen Client und Server basiert. Man kann zwar alle paar Sekunden einen sogenannten Server-Poll einleiten, d.h. beim Webserver nachfragen, ob neue Chat-Beiträge bzw. Direktnachrichten vorhanden sind. Allerdings lässt sich damit nicht die Geschwindigkeit und Zuverlässigkeit erreichen, wie man sie von echten Chat-Programmen gewohnt ist.

Unter dem Schlagwort *Comet* werden allerdings Techniken wie *Web Sockets* und *Server-sent Events* entwickelt, die diese

Beschränkungen von HTTP zu umgehen versuchen. Auch wenn es bereits beeindruckende Praxisanwendungen gibt, so kommen diese nicht ohne aufwändige Tricks aus und haben mit grundlegenden Problemen zu kämpfen.

XMLHttpRequest – die JavaScript-Technik hinter Ajax

Nachdem wir einige Anwendungsfälle betrachtet haben, soll es nun zur Sache gehen: Wie wird Ajax in JavaScript konkret angewendet?

Hintergrund-Serverkommunikation wird in erster Linie mit einer JavaScript-Technik umgesetzt: dem**`XMLHttpRequest`-Objekt**. Dies ist ursprünglich eine proprietäre Erfindung von Microsoft für den Internet Explorer. Die Erzeugung eines `XMLHttpRequest`-Objekt war zunächst an ActiveX gekoppelt, eine weitere Microsoft-Technik. Mithilfe dieses Objektes werden HTTP-Anfragen gestartet und die Server-Antwort ausgelesen.

Andere Browserhersteller erkannten die Möglichkeiten von `XMLHttpRequest` und übernahmen diese Technik - allerdings ohne ActiveX. Mittlerweile kennen alle großen JavaScript-fähigen Browser das Objekt `window.XMLHttpRequest`. Ab Version 7 des Internet Explorers ist dieses globale Objekt ebenfalls verfügbar, bei älteren Versionen muss der Umweg über ActiveX genommen werden.

Asynchronität und Event-Basierung bei XMLHttpRequest

Der Clou an `XMLHttpRequest` ist, dass es eine Server-Anfrage standardmäßig *asynchron*, d.h. im Hintergrund absendet. Die Server-Antwort wird dann durch Ereignis-Behandlung verarbeitet. Das bedeutet, dass ein Script die Anfrage auslöst und eine angegebene Event-Handler-Funktion aufgerufen wird, sobald sich der Status der Anfrage ändert und schließlich die Antwort eintrifft.

Diese Ereignis-basierte Verarbeitung hat folgenden Sinn: Der Browser friert an der Stelle, wo die Anfrage abgesendet wird, nicht ein und stoppt die Ausführung von JavaScript, bis die Antwort eingetroffen ist. Sondern der Browser kommt zur Ruhe, kann andere Scriptteile ausführen und sogar weitere Server-Anfragen starten. Erst dadurch ist die schnelle und unterbrechungsfreie Reaktion auf Benutzereingaben möglich, die für Ajax-Anwendungen typisch ist.

Wenn Sie noch nicht mit Event-Handling in JavaScript vertraut sind, dann wird Sie dieses Modell erst einmal verwirren. Denn die JavaScript-Funktion, die das XMLHttpRequest erzeugt und die HTTP-Anfrage absendet, kann nicht gleichzeitig die Server-Antwort verarbeiten. Diese Aufgabe muss eine weitere Funktion übernehmen. Wie das konkret aussieht, werden wir später sehen.

In den meisten Fällen sollten Sie asynchrones XMLHttpRequest wählen. Es soll allerdings nicht verschwiegen werden, dass auch synchrones XMLHttpRequest möglich ist. Dies arbeitet nicht Event-basiert, sondern hält die JavaScript-Ausführung vom Zeitpunkt des Absendens der Anfrage bis zum Zeitpunkt des vollständigen Empfangs der Antwort an. Das bedeutet, dass die JavaScript-Anweisung, die auf `xhr.send()` folgt (siehe unten) direkt Zugriff auf die Server-Antwort hat.

Server-Anfrage mit XMLHttpRequest absenden

Die Absenden einer Anfrage mittels XMLHttpRequest umfasst vier grundlegende Schritte:

Das Erzeugen eines XMLHttpRequest-Objektes

```
var xhr = new XMLHttpRequest();
```

Eine neue Instanz wird erzeugt, indem der Konstruktor `XMLHttpRequest` mit dem Schlüsselwort`new` aufgerufen wird. Das zurückgelieferte Anfrage-Objekt wird hier in einer Variable namens `xhr`gespeichert.

Für ältere Internet Explorer ist wie gesagt eine andere Schreibweise nötig, welche ein ActiveX-Objekt erstellt. ...

Um browserübergreifend zu arbeiten, benötigen wir eine Fähigkeitenweiche mit einer Objekt-Erkennung. Wenn window.XMLHttpRequest zur Verfügung steht, wird diese Objekt benutzt, andernfalls wird versucht, ein ActiveX-Objekt zu erzeugen. ...

Registrieren einer Handler-Funktion für das `readystate`-Ereignis (bei asynchronen Anfragen)

```
xhr.onreadystatechange = xhrReadyStateHandler;
```

Sie müssen eine Funktion angeben, die immer dann aufgerufen wird, wenn sich das Status der Server-Anfrage ändert. Das bewerkstelligt die Zuweisung einer Funktion an die Eigenschaft`onreadystatechange`. Dieses Schema gleich dem traditionellen Event-Handling. Der Aufbau der`readyState`-Handler-Funktion wird weiter unten beschrieben.

Diese Anweisung ist nur bei asynchronen Anfragen nötig.

Festlegen der Anfrage-Methode, der Anfrage-Adresse sowie des Arbeitsmodus

```
xhr.open("GET", "beispiel.html", true);
```

Mit dem Aufruf der `open`-Methode des Anfrage-Objektes geben Sie drei zentrale Daten an.

1. Im ersten Parameter die HTTP-Anfragemethode als String. Üblich sind `"GET"`, `"POST"` und`"HEAD"`.
2. Im zweiten Parameter die Adresse (URI) als String, an die die Anfrage gesendet werden soll. Im Beispiel wird davon ausgegangen, dass unter demselben Pfad wie das HTML-Dokument, das die Server-Anfrage startet, eine Datei namens *beispiel.html* existiert.
3. Der dritte Parameter legt schließlich fest, ob die Anfrage synchron oder asynchron abgesendet werden soll. Übergeben Sie `true` für eine asynchrone Abfrage oder `false` für eine synchrone Anfrage.

Im obigen Beispielcode wird eine asynchrone GET-Anfrage an die Adresse *beispiel.html*abgesendet.

Anfrage starten und gegebenenfalls POST-Daten angeben

```
xhr.send(null);
```

Um die Anfrage zu starten, rufen Sie die `send`-Methode auf. Im Fall einer GET-Anfrage brauchen Sie keine Parameter angeben - oder nur `null` wie im Beispiel. Im Falle einer POST-Anfrage übergeben Sie den sogenannten Anfragekörper (engl. *POST body*). Dieser String wird dann an den Server übermittelt. Die Übertragung von Daten werden wir weiter unten genauer besprechen.

Server-Antwort beim XMLHttpRequest verarbeiten

Vor dem Absenden eines asynchronen XMLHttpRequest haben wir eine Handler-Funktion namens`xhrReadyStateHandler` registriert. Diese wird beispielhaft wie folgt notiert:

```
function xhrReadyStateHandler () {
   …
}
```

Anfrage-Status: `readyState`

Die Funktion wird immer dann ausgeführt, wenn sich der Status der Anfrage ändert (der sogenannte *ready state*, auf deutsch in etwa Fortschrittsstatus). Jede Anfrage durchläuft eine Reihe von fünf Phasen, die jeweils durch eine Nummer von 0 bis 4 identifiziert werden.

Der gegenwärtigen Status lässt sich über die Eigenschaft `readyState` des XMLHttpRequest-Objektes abfragen. Diese Eigenschaft enthält eine Nummer gemäß der folgenden Tabelle.

Status-Nummer	Anfrage-Zustand
0	XMLHttp-Request wurde erzeugt, aber die `open`-Methode noch nicht aufgerufen
1	`open` wurde aufgerufen, aber noch nicht `send`
2	`send` wurde aufgerufen und von der Server-Antwort ist bereits der Statuscode und die Antwort-Kopfzeilen (Header) eingetroffen
3	Die Antwortdaten werden übertragen und sind schon teilweise verfügbar

4	Die Antwortdaten wurden komplett übertragen

In den meisten Fällen ist nur der Status 4 interessant, denn dann können Sie auf die vollständige Server-Antwort zugreifen. Da der `readyState`-Handler für jeden Status aufgerufen wird, fragen wir darin ab, ob die `readyState`-Eigenschaft den Wert 4 erreicht hat. Wenn das nicht der Fall ist, brechen wir die Funktion vorzeitig mit `return` ab. Der restliche JavaScript-Code der Funktion wird dann nur ausgeführt, wenn der Zugriff auf die Server-Antwort möglich ist.

```
function xhrReadyStateHandler () {
   if (xhr.readyState != 4) {
      return;
   }
   // Server-Antwort ist eingetroffen!
}
```

HTTP-Statuscode: `responseCode`

Alleine das Eintreffen der Server-Antwort bedeutet nun nicht, dass der Server die Anfrage fehlerfrei verarbeiten konnte. Ausschlaggebend dafür ist der *HTTP-Statuscode* der Server-Antwort. Dies ist eine vierstellige Zahl. Im Erfolgsfalle lautet der Code `200`, im Fehlerfalle z.B. `404` für »Nicht gefunden« oder `500` für »Server-interner Fehler«. Nur im Erfolgsfalle können Sie damit rechnen, dass der Server die gegebenenfalls übersandten Daten entgegen genommen hat und die Server-Antwort die gewünschten Daten enthält.

Der HTTP-Statuscode der Antwort lässt sich über die Eigenschaft `responseCode` des jeweiligen XMLHttpRequest-Objektes auslesen. Der `readyState`-Handler wird daher um eine `if`-Anweisung ergänzt, die den `responseCode` mit `200` vergleicht. `200` bedeutet wie gesagt »Alles in Ordnung«.

```
function xhrReadyStateHandler () {
   if (xhr.readyState != 4) {
      return;
   }
   // Server-Antwort ist eingetroffen!
   if (xhr.responseCode == 200) {
      // Server-Antwort in Ordnung
   }
}
```

Im Fehlerfall passiert beim obigen Beispiel nichts – das ist natürlich nicht optimal. Denn selbst wenn der Server nicht mit den gewünschten Daten antwortet, so antwortet er zumeist mit einer Fehlermeldung. Das folgende Beispiel enthält in einem `else`-Zweig eine sehr einfache Fehlerbehandlung:

```
function xhrReadyStateHandler () {
   if (xhr.readyState != 4) {
      return;
   }
   if (xhr.responseCode == 200) {
      // Server-Antwort in Ordnung
   } else {
      alert("Es ist ein Fehler beim Laden aufgetreten:\n" + xhr.responseCode);
   }
}
```

Diese kann Ihnen als Seitenbetreiber helfen, den Fehler zu beheben – für Ihre Seitenbesucher ist sie allerdings oftmals wenig hilfreich.

Zugriff auf die Antwortdaten: `responseText` und `responseXML`

Nachdem der Anfrage-Status sowie der HTTP-Antwortcode abgefragt wurde, kann endlich auf die tatsächlichen Antwortdaten zugegriffen werden. Dazu stehen zwei Eigenschaften des XMLHttpRequest-Objekts zur

Verfügung: `responseText` und `responseXML`.

`responseText`

`responseText` enthält die Server-Antwort als String. Das Auslesen dieser Eigenschaft ist der Standardweg und die folgenden Beispiele werden diese Eigenschaft verwenden.

`responseXML`

Wenn es sich bei der Server-Antwort um ein XML-Dokument handelt, erlaubt die Eigenschaft`responseXML` Zugriff auf das DOM des XML-Dokumentes. Vorausgesetzt ist, dass der Server das Dokument mit einem entsprechenden Inhaltstyp (MIME-Typ) sendet. Dieser lautet ist üblicherweise`application/xml`. Nur dann verarbeitet der Browser die Server-Antwort automatisch mit seinem XML-Parser und stellt den DOM-Zugriff bereit.

`responseXML` liefert den `Document`-Knoten des DOM-Knotenbaums. Von dort aus stehen Ihnen alle Möglichkeiten des W3C-Core-DOM zur Verfügung. Auf das Wurzelelement können Sie beispielsweise über `xhr.responseXML.documentElement` zugreifen, eine Liste aller Elemente eines Typs bekommen Sie mit den Methoden `xhr.responseXMLgetElementsByTagName()` bzw.`xhr.responseXMLgetElementsByTagNameNS()` abfragen.

Server-Antwort aus `responseText` ins Dokument schreiben

Eine häufige Aufgabe ist es, die Server-Antwort ins aktuelle Dokument einzufügen. Das bekannte Beispiel erweitern wir um eine Zuweisung, die die Antwortdaten in ein Element hineinlädt.

```
function xhrReadyStateHandler () {
   if (xhr.readyState != 4) {
      return;
   }
   if (xhr.responseCode == 200) {
      document.getElementById("zielelement").innerHTML = xhr.responseText;
   }
}
```

Mittels `getElementById` sprechen wir ein vorhandenes Element im Dokument an. Der String mit den Antwortdaten wird dessen Eigenschaft `innerHTML` zugewiesen. Die Antwortdaten werden damit als HTML-Codeschnipsel interpretiert und werden als neuer Inhalt des angesprochenen Elements eingebunden.

Übrigens ist es auch möglich, `responseText` zu nutzen, wenn der Server nicht mit dem HTTP-Statuscode `200` geantwortet hat – in diesem Fall können Sie `responseText` auslesen, um eine eventuelle Fehlermeldung des Servers auszugeben.

Vollständiges XMLHttpRequest-Beispiel

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html><head>
<title>XMLHttpRequest-Beispiel</title>
<script type="text/javascript">
function sendXhr () {
   var xhr;
   if (window.XMLHttpRequest) {
      xhr = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
      xhr = new ActiveXObject('MSXML2.XMLHTTP');
   } else {
      return false;
   }
   xhr.onreadystatechange = xhrReadyStateHandler;
   xhr.open("GET", "beispiel.html", true);
   xhr.send(null);
}
```

```
function xhrReadyStateHandler () {
   if (xhr.readyState != 4) {
      return;
   }
   if (xhr.responseCode == 200) {
      document.getElementById("zielelement").innerHTML = xhr.responseText;
   }
}
</script>
</head><body>
<p><a href="javascript:sendXhr()">XMLHttpRequest starten</a></p>
<p id="zielelement">Hier erscheint die Server-Antwort.</p>
</body></html>
```

...

Datenübertragung mit XMLHttpRequest

Die Vielseitigkeit von XMLHttpRequest rührt daher, das Sie nicht nur Daten vom Server nachladen können, sondern auch Daten aus JavaScript heraus an den Server übermitteln können. Beispielweise lässt sich mit JavaScript der Wert eines Formularelements im Dokument auslesen und per XMLHttpRequest an den Server übertragen. Dieser interpretiert den Wert beispielsweise als Suchbegriff und liefert entsprechende Suchergebnisse zurück.

Wie die Datenübertragung genau abläuft, hängt von der Anfragemethode ab. Wir werden die zwei wichtigsten betrachten: `GET` und `POST`.

`GET` und der Query String

Bei GET werden die zu übertragenden Daten in die Adresse (URI) einmontiert, und zwar in den Teil der URI, der *Query String* (Abfrage-Zeichenkette) genannt wird. Dieser ist von der eigentlichen URI mit einem Fragezeichen (`?`) getrennt. Eine relative Adresse mit Query String sieht von Aufbau her so aus:

```
beispiel.php?querystring
```

Üblicherweise setzt sich der Query String aus Name-Wert-Paaren zusammen. Name und Wert werden durch ein Gleichheitszeichen (`=`) getrennt, die Paare durch ein kaufmännisches Und-Zeichen (`&`). Schematisch ist der Query String folglich so aufgebaut: `name1=wert1&name2=wert2&name3=wert3`usw. (Je nach verwendeter Server-Software ist auch ein anderer Aufbau des Query Strings denkbar. Dies ist jedoch der am weitesten verbreitete.)

Wenn wir beispielsweise die drei Informationen Vorname: Alexandra, Nachname: Schmidt und Wohnort: Hamburg übertragen wollen, so könnte der Query String lauten:

```
vorname=Alexandra&nachname=Schmidt&wohnort=Hamburg
```

Um diese Daten per `GET` zu übertragen, werden sie in die relative Adresse einmontiert:

```
beispiel.php?vorname=Alexandra&nachname=Schmidt&wohnort=Hamburg
```

Diese Adresse könnten wir bereits beim Starten eines XMLHttpRequest nutzen, d.h. sie als Parameter an die `open`-Methode übergeben:

```
xhr.open("GET", "beispiel.php?vorname=Alexandra&nachname=Schmidt&wohnort=Hamburg", true);
```

Spannend wird es aber erst, wenn es gilt, diese Adresse samt Query String dynamisch mit variablen Daten zusammenzusetzen, die z.B. der Benutzer eingegeben hat.

Nehmen wir an, es gibt im Dokument folgendes Formular mit den drei Feldern für Vorname, Nachname und Wohnort:

```
<form id="formular" action="beispiel.php">
<p><label>Vorname: <input type="text" name="vorname"></label></p>
<p><label>Nachname: <input type="text" name="nachname"></label></p>
<p><label>Wohnort: <input type="text" name="wohnort"></label></p>
<p><input type="submit" value="Absenden"></p>
</form>
```

Wenn JavaScript aktiviert ist, soll das Formular im Hintergrund per XMLHttpRequest abgesendet werden. Andernfalls erfolgt ein normales Absenden samt Auswechseln des gegenwärtig angezeigten Dokuments

Mittels traditionellem Event-Handling verknüpfen wir JavaScript-Logik mit dem Absenden des Formulares. Dazu nutzen wir das übliche mehrschrittige Schema:

```
window.onload = initFormularXHR;

function initFormularXHR () {
   document.getElementById("formular").onsubmit = formularSubmit;
}

function formularSubmit () {
   // Formular mit XMLHttpRequest absenden
}
```

Die Handler-Funktion `formularSubmit` muss nun folgendes leisten:

1. Das Formular über das DOM ansprechen
2. Den Query String mit den Formulareingaben zusammensetzen (man spricht von einer**Serialisierung** des Formulars)
3. Einen XMLHttpRequest starten mit dem erzeugten Query String
4. Das normale Absenden des Formulars unterbrechen, indem die Standardaktion abgebrochen wird

Am kompliziertesten ist der zweite Schritt, deshalb betrachten wir erst einmal das Gerüst mit den restlichen Aufgaben.

```
function formularSubmit () {
}
```

...

Helferfunktion für den zweiten Schritt, die Serialisierung des Formulars

1. Um den Query String zusammenzusetzen, werden die Felder des Formulars durchlaufen
2. Für jedes Feld dessen Namen und Wert auslesen und ein entsprechendes Paar erzeugen (`name=wert`)
3. Dieses Paar an einen String anhängen, in dem der Query String zusammengesetzt wird

POST und der POST-Anfragekörper

Helferfunktion für XMLHttpRequest

Alternativtechniken zur Serverkommunikation

Neben `XMLHttpRequest` gibt es weitere JavaScript-Techniken, um im Hintergrund Daten mit dem Webserver auszutauschen. Strenggenommen wurde Ajax schon lange praktiziert, bevor sich`XMLHttpRequest` breiter Browserunterstützung erfreute. Aber erst

mit `XMLHttpRequest` gelang Ajax der Durchbruch, weshalb beide Begriffe oft synonym verwendet werden. Dies allerdings zu Unrecht: Zwar ist `XMLHttpRequest` die vielseitigste, aber beileibe nicht immer die einfachste und geeignetste Methode, um Informationen mit dem Webserver auszutauschen. Alternativen sind unter anderem:

Iframes

In eingebettete Frames (Iframes) können Daten im Hintergrund geladen werden oder sie dienen als Ziel-Frame für Formulare. Ein Script im einbettenden Elterndokument kann auf diesen Frame und seine Inhalte zugreifen. Da der Website-Besucher nichts von diesen Vorgängen im Hintergrund mitbekommen muss, kann der Iframe kurzerhand mit CSS versteckt werden.

Bildobjekte

Schon seit den Anfängen von JavaScript existiert die Möglichkeit, per JavaScript Bildobjekte (`Image`-Objekte) zu erzeugen, um Grafiken in den Browsercache zu laden:

```
var bild = new Image();
bild.src = "beispiel.png";
```

Diese `Image`-Objekte lassen sich jedoch zweckentfremden, um eine Server-Anfrage (`GET`-Anfrage) zu starten. Es ist dabei nicht wichtig, ob der Server auch tatsächlich eine Bilddatei (oder überhaupt etwas) zurücksendet. Diese im Vergleich zu `XMLHttpRequest` sehr einfache Methode eignet sich für die Server-Aktionen, in denen man dem Server per `GET`-Anfrage etwas mitteilt, aber keine Antwort oder Bestätigung erwartet.

`script`-Elemente

Per JavaScript erzeugte `script`-Elemente, die Scripte selbst von fremden Servern einbinden können, die Daten in mit JavaScript direkt zugänglichen Formaten bereitstellen.

Besonderheiten: Cross-Site JSON und JSONP

Übertragungsformate

HTML / Nur-Text

...

JSON

...

```
[
   {
      "name" : "Harry S. Truman",
      "partei" : "Demokraten",
      "beginn" : 1945,
      "ende" : 1953
   },
   {
      "name" : "Dwight D. Eisenhower"
      "partei" : "Republikaner",
      "beginn" : 1953,
      "ende" : 1961
   },
   {
      "name" : "John F. Kennedy"
      "partei" : "Demokraten",
      "beginn" : 1961,
```

```
      "ende" : 1963
   }
]
```

...

XML

`XMLHttpRequest` hat seinen Namen daher, dass es ursprünglich dazu gedacht war, XML-Dokumente vom Server herunterzuladen. JavaScript kann anschließend mit verschiedenen Schnittstellen mit dem Dokument arbeiten. Das funktioniert zwar vorzüglich, allerdings ist das Extrahieren von Daten aus einem XML-Dokument über die DOM-Schnittstelle umständlich. In den meisten Fällen sollen die Daten nämlich im bestehenden HTML-Dokument angezeigt werden, sodass die XML-Daten in eine HTML-Struktur umgewandelt werden müssen.

Wenn Sie strukturierte Daten per XML übertragen wollen, müssen Sie ein eigenes Auszeichnungssprache nach den Regeln von XML entwerfen. Sie denken sich gewisse Elemente und Attribute und die grammatischen Regeln aus, wie diese zusammengesetzt werden können. Mithilfe dieser Datenstrukturen werden die zu übertragenden Informationen geordnet. Das zugehörige JavaScript kann diese Elemente ansprechen und die Werte der enthaltenen Textknoten bzw. Attribute auslesen.

Auch wenn Sie diese XML-Sprache nicht unbedingt in einer formalen Dokumentyp-Definition (DTD) niederschreiben müssen, ist das Entwerfen eines passenden XML-Derivates für für Ajax vergleichsweise aufwändig. Das Auslesen der Informationen im JavaScript erfordert nicht nur Kenntnisse des XML-DOM, sondern ist mit ziemlicher Schreibarbeit verbunden.

Wenn die serverseitigen Programme, die Ihre Website betreiben, bereits ein XML-Format als Speicher- oder Austauschformat verwendet und die damit ausgezeichneten Daten zum Client weitergeschickt werden sollen, bietet es sich auch für Ajax an. Andernfalls ist XML oftmals kompliziert und überdimensioniert.

...

```
<?xml version="1.0" encoding="UTF-8" ?>
<us-präsidenten>
   <präsident>
      <name>Harry S. Truman</name>
      <partei>Demokraten</partei>
      <beginn>1945</beginn>
      <ende>1953</ende>
   </präsident>
   <präsident>
      <name>Dwight D. Eisenhower</name>
      <partei>Republikaner</partei>
      <beginn>1953</beginn>
      <ende>1961</ende>
   </präsident>
   <präsident>
      <name>John F. Kennedy</name>
      <partei>Demokraten</partei>
      <beginn>1961</beginn>
      <ende>1963</ende>
   </präsident>
</us-präsidenten>
```

Ajax und Sicherheit

Same-Origin-Policy bei XMLHttpRequest

XMLHttpRequest unterliegt dem grundlegenden Sicherheitskonzept der Same-Origin-Policy. Das bedeutet, sie können mit XMLHttpRequest nur HTTP-Anfragen an den Webserver absenden, auf dem das Dokument liegt, in dessen Kontext das JavaScript ausgeführt wird.

Wenn Sie von einem Dokument auf *example.org* aus eine Anfrage an eine fremden Domain, z.B.*example.net* senden, dann wird der Browser das JavaScript mit einem Ausnahmefehler abbrechen.

```
// Script auf example.org
xhr.open("GET", "http://example.net/", true); // Fehler!
```

Sie können also nicht ohne weiteres HTTP-Anfragen an beliebige Adressen senden bzw. beliebige Adressen im Web auslesen – das verhindert die Same-Origin-Policy. Es gibt jedoch zahlreiche Möglichkeiten, mit JavaScript auf externe Dienste und Datenquellen im Web zuzugreifen und sogenannte*Mashups* zu erstellen. Diese verwenden nicht klassisches XMLHttpRequest, sondern nutzen alternative Techniken, bei denen andere Sicherheitsmodelle als Same-Origin-Policy greifen.

Cross-Site Ajax mit HTTP Access Control

Cross-Site, *Cross-Origin* oder auch *Cross-Domain Ajax* bezeichnet die Möglichkeit, eine HTTP-Anfrage an eine fremde Domain abzusenden und auf die Server-Antwort zuzugreifen. Derzeit etwickelt das Standardisierungsgremium W3C einen Ansatz, der Schnittstellen wie XMLHttpRequest einen domainübergreifenden Zugriff ermöglichen soll. Der Entwurf für die technische Spezifikation firmiert derzeit unter dem Namen Cross-Origin Resource Sharing (engl. für Herkunft-übergreifender Zugriff auf Ressourcen), ist aber bekannter unter dem Namen **HTTP Access Control** (engl. für HTTP-Zugriffskontrolle).

Mit diesem Modell lassen sich Ausnahmen von der Same-Origin-Policy festlegen, und zwar nach einem **Einverständnis-Modell**: Eine Website erlaubt einer bestimmten anderen Website, auf sie zuzugreifen. Gemäß dem obigen Beispiel kann die Zieldomain `example.net` einwilligen, dass Anfragen von Ursprungsdomain `example.org` erlaubt sind.

Diese Erlaubnis wird in einer bestimmten HTTP-Kopfzeile (Header) erteilt, die der Webserver unter`example.net` bei der Beantwortung einer Anfrage mitliefert. Um beispielsweise den domainübergreifenden Zugriff ausschließlich von Dokumenten auf `http://example.org` zu erlauben, sendet der Webserver die Kopfzeile:

```
Access-Control-Allow-Origin: http://example.org
```

Die Überprüfung der Erlaubnis nimmt der Browser vor: Wenn ein JavaScript von `example.org`eine Anfrage an `example.net` sendet, so prüft der Browser, ob die Antwort von `example.net` die besagten HTTP-Kopfzeile enthält. Falls ja, so kann das JavaScript die Serverantwort wie gewohnt verarbeiten, andernfalls wird das Script mit einem Ausnahmefehler abgebrochen. (???)

Dies ist nur ein vereinfachtes Beispiel, um einfache Anfragen zu erlauben. Die besagte Spezifikation sieht viel feinere Zugriffskontrolle vor. An dieser soll lediglich das Grundprinzip ersichtlich werden.

In der Praxis gibt es derzeit zwei Umsetzungen der W3C-Spezifikation: Mozilla Firefox erlaubt ab Version 3.5 den domainübergreifenden Zugriff mithilfe eines normalen `XMLHttpRequest`-Objekts. Der Internet Explorer ab Version 8 hat hingegen ein eigenes Objekt nur solche Zugriffe erfunden:XDomainRequest. Dessen Bedienung ähnelt der von XMLHttpRequest stark, es gibt im Detail allerdings einige Unterschiede, Eigenheiten und Verbesserungen.

Da beide Techniken noch recht neu und noch nicht breit unterstützt sind, sei hier nur am Rande darauf hingewiesen. ...

Wie gesagt setzen diese Techniken die Same-Origin-Policy nicht vollständig außer Kraft, sondern ergänzen das Verbot des domainübergreifenden Zugriffs durch eine Ausnahme: Der Webserver auf der Zieldomain muss so konfiguriert sein, dass er sein Einverständnis zu dieser fremden Anfrage gibt.

JavaScript: Bibliotheken und Frameworks

1. Notizen
2. Einführung

3. Vor- und Nachteile von Frameworks
4. Eigene Bibliotheken zusammenstellen
5. Verbreitete Frameworks

Notizen

- JavaScript-Doku
- JavaScript-Bibliotheken und die jüngere JavaScript-Geschichte
- Sieben Thesen zum gegenwärtigen JavaScript-Gebrauch
- Organisation von JavaScripten: Ausblick auf JavaScript-Frameworks

Einführung

Wie bei jeder Programmiersprache finden sich unzählige Fertigscripte und Codeschnipsel im Netz, die Sie in Ihren Programmen verwenden können und die spezifische Aufgaben lösen. Um solche Fertig- und Helferscripte soll es hier nicht gehen, sondern um Bibliotheken, die den Programmierstil grundlegend ändern können.

Die meisten Web-Programmiersprachen bringen die wichtigsten Funktionen schon mit und es gibt einen relativ festen Rahmen, der zentral weiterentwickelt wird. In JavaScript hingegen gibt es verschiedene Schnittstellen unterschiedlicher Herkunft, die zum Teil alles andere als einfach und intuitiv sind. Es gibt einflussreiche proprietäre Erweiterungen und selbst der Sprachkern ist in den verschiedenen Browsern unterschiedlich umgesetzt (siehe). Für Standardaufgaben wie das Event-Handling gibt es in JavaScript kein verlässliches Fundament.

Sogenannte **Frameworks** (englisch Rahmenwerk, Grundstruktur) kaschieren diese chaotische Situation, indem sie eine Abstraktionschicht über JavaScript legen. Ziel ist es, dass grundlegende Aufgaben des *DOM Scripting* nicht immer wieder von Hand mit viel Aufwand erledigt werden müssen. Anstatt z.B. direkt mit dem DOM zu arbeiten, führen Frameworks einige Objekte und Methoden als Abstraktionsschicht ein. Diese sind einfacher und intuitiver zu bedienen und nehmen dem Webautor einen Großteil der Arbeit ab.

Heutige Frameworks setzen auf Vereinfachungen unter anderem in folgenden Bereichen:

1. **Arbeiten mit dem DOM-Elementenbaum:**
    1. Effizientes Ansprechen von Elementen im Dokument z.B. über CSS-artige Selektoren
    2. Durchlaufen und Durchsuchen von Elementlisten nach bestimmten Kriterien
    3. Einfaches Ändern des DOM-Baumes, Einfügen von neuen Elementen und Textinhalten
    4. Zuverlässiges Auslesen und Setzen von Attributen
2. **Event-Handling:**
    1. Aufrufen von Initialisierungs-Funktionen, sobald der DOM-Baum verfügbar ist, um dem Dokument die gewünschte Interaktivität hinzuzufügen
    2. Registrieren und Entfernen von Event-Handlern sowie die Kontrolle des Event-Flusses
    3. Browserübergreifender Zugriff auf Event-Eigenschaften wie die Mausposition und gedrückte Tasten
3. **Darstellung abfragen und beeinflussen:**
    1. Auslesen und Ändern der CSS-Eigenschaften von Elementen
    2. Hinzufügen und Entfernen von Klassen
    3. Animationen und Effekte, Wiederholungen und Verzögerungen
    4. Zugriff auf die Eigenschaften des sogenannten *Viewports* (der rechteckige Raum, in dem die Webseite dargestellt wird), der Position sowie der Ausmaße eines Elements

4. Vereinfachtes Arbeiten mit **Formularen**
5. **Ajax**: Kommunikation mit dem Webserver, um Daten zu übertragen oder nachzuladen, ohne das Dokument zu wechseln (`XMLHttpRequest`)
6. Strukturierte und **objektorientierte Programmierung**, Modulverwaltung

Dies ist nur eine Auswahl der wichtigsten Bereiche, sodass Sie einen Einblick bekommen, an welchen Stellen die meisten Frameworks Vereinfachungen anbieten. Darüber hinaus bieten viele Frameworks freilich noch weitere Funktionen an, z.B. Bedienelemente (englisch *Controls* oder *Widgets*) für JavaScript-Webanwendungen.

Vor- und Nachteile von Frameworks

In letzter Zeit ist gleichsam ein Hype um Frameworks zu verzeichnen. Ihr Einsatz ist oft mit überzogenen Erwartungen und Missverständnissen verbunden. Diese sollen im Folgenden kritisch beleuchtet werden.

Frameworks sollten hier weder verteufelt noch vergöttert werden. Ihre Anwendung ist nämlich ein zweischneidiges Schwert:

Frameworks helfen Anfängern, setzen aber JavaScript-Grundkenntnisse voraus

Frameworks legen ein Abstraktionsschicht über die Browserunterschiede, vereinheitlichen den Flickenteppich von schwer zu bedienenden Schnittstellen und bieten Helferfunktionen an. Das kann Webautoren viel Arbeit abnehmen und vor allem Unerfahrenen dabei helfen, sich in der unübersichtlichen JavaScript-Welt zurechtzufinden.

Sie sollten jedoch nicht darauf hoffen, dass Frameworks das Schreiben von JavaScript ohne JavaScript-Kenntnisse ermöglichen! Frameworks nehmen Ihnen nicht das Verständnis der Grundlagen ab, sondern setzen diese stillschweigend voraus. Frameworks können Ihnen in gewissen Punkten helfen und Vorgaben machen, an vielen Stellen sind sie jedoch weiterhin auf Ihre eigenen Programmierfähigkeiten angewiesen.

Abstraktion hat Vor- und Nachteile

Die Abstraktionsschicht verbirgt die internen Vorgänge und gibt vor, einen schnellen Einstieg in eine schwierige Materie zu ermöglichen. Es ist in vielen Fällen jedoch unverzichtbar, die interne Arbeitsweise zu kennen. Hier gilt: Wenn Sie die Aufgaben schon einmal ohne Framework von Hand gelöst haben bzw. die Lösungsansätze kennen, stehen Sie nicht im Regen, wenn die Abstraktion in der Praxis nicht mehr greifen sollte.

Frameworks stecken voller Know-How und eine effiziente Anwendung erfordert Profiwissen

Frameworks setzen Programmiertechniken ein, die dem gemeinen JavaScript-Programmierer meist unbekannt sind. Die ausgiebige Nutzung von Frameworks läuft zudem immer wieder darauf hinaus, dass Sie in deren Code schauen müssen, um gewisses Verhalten zu verstehen. Das Nachvollziehen dieses anspruchsvollen Codes erfordert wiederum fortgeschrittenes JavaScript-Wissen.

Dass die heutigen Frameworks in einer bestimmten Weise aufgebaut sind, ist das Resultat eines langen Entwicklungsprozesses. Wenn Sie die ausgeklügelten Techniken der Frameworks einsetzen möchten, kommen Sie nicht umhin, diese Entwicklung nachzuvollziehen.

Frameworks *können* den Stil entscheidend verbessern

Mit verschiedenen Frameworks ist ein hochentwickelter Programmierstil möglich. Sie betonen vernachlässigte JavaScript-Fähigkeiten wie die funktionale und objektorientierte Programmierung. Zudem bringen sie bewährte Konzepte aus anderen Sprachen in die JavaScript-Welt.

Auch hier gilt: Um diese Möglichkeiten auszureizen, müssen Sie sich intensiv mit den dahinterstehenden Konzepten auseinandersetzen. Alleine durch den Einsatz eines Frameworks wird ihr Code nicht automatisch eleganter, strukturierter oder kompatibler - Frameworks können Ihnen aber entscheidende Hinweise geben.

Dokumentationen sind größtenteils unzureichend

Sie sollten die Herausforderung nicht unterschätzen, die Schnittstelle eines Frameworks zu überblicken und zu verstehen. Die verbreiteten Frameworks bieten leider wenig gute Dokumentation und Anleitungen, die Ihnen die jeweiligen Grundideen vermitteln.

Ausgehend von diesen Beobachtungen sei Ihnen folgender Umgang mit Frameworks empfohlen:

1. Lernen Sie auf jeden Fall die **Grundlagen von JavaScript**! Dabei werden Sie die Schwierigkeiten der Praxis kennenlernen und eine Eindruck davon bekommen, wo Frameworks ansetzen.

2. Entwickeln Sie zunächst **eigene kleine Bibliotheken**, indem Sie ständig wiederholte Vorgänge in Helferfunktionen auslagern. Frameworks sind der Versuch, solch lose Helferfunktionen zu einer neuen, einheitlichen Schnittstelle zusammenzufassen.

3. Lernen Sie eine verbreitetes Framework und dessen typischen Programmiertechniken kennen. Verschaffen Sie sich einen Überblick über den grundlegenden Funktionsumfang, sodass Sie bekannte Aufgaben mithilfe des Frameworks schneller umsetzen können. Entwickeln Sie anfangs kleine Scripte mithilfe des Frameworks und versuchen Sie, dabei die Möglichkeiten des Frameworks auszureizen.

4. Wenn Sie tiefer in die JavaScript-Programmierung einsteigen wollen, informieren Sie sich über die Konzepte der verschiedenen Frameworks, ihre Unterschiede und Grenzen.

Eigene Bibliotheken zusammenstellen

Bei den verbreiteten Frameworks ist zu beobachten, dass viele Anwender den Funktionsumfang nicht ausnutzen, sondern gerade einmal von drei oder vier gesonderten Funktionen Gebrauch machen. Die begrüßenswerten Innovationen haben die Anwender größtenteils noch nicht erreicht. Das liegt zum einen an den besagten fehlenden Dokumentationen, zum anderen an der Komplexität und fehlenden Modularisierbarkeit. Heraus kommt ein Code, bei dem Framework-Logik und herkömmlichen Lösungsweisen vermischt werden. Dieses Kauderwelsch ist inkompatibel, schwer verständlich und schlecht wartbar.

Unklare Sache von: mschaefer

zuviel Gelaber

Es ist deshalb nicht immer klug, ein vorgefertigtes, umfangreiches Framework zu nehmen, von dem der größte Teil unverstanden ist. Setzen Sie diejenigen Hilfsmittel ein, die sie verstehen und beherrschen. Sie können Sie Ihre JavaScript-Programmierung bereits entscheidend vereinfachen, indem Sie **punktuell Helferscripte einsetzen**. Diese Scripte helfen dabei, häufige Aufgaben umzusetzen und gängige Probleme zu lösen. In der Regel sind sie leichter verständlicher und erfordern wenig Einarbeitung.

Im Folgenden werden Bausteine kurz vorgestellt, aus denen Sie sich eine eigene Bibliothek zusammenstellen können. Diese Vorstellung ist auch gleichzeitig ein Einstieg in die Funktionsweise der »großen« Frameworks.

»Scripting Essentials« von Dan Webb

Diese Zusammenstellung von Helferfunktionen ist ein gutes Beispiel dafür, wie bereits ein paar gezielte Funktionen die Programmierung vereinfachen und verbessern können. Dan Webb nennt sie seine Scripting Essentials [en]. Code herunterladen. Das Script umfasst folgende Teile:

Verbessertes Arbeiten mit Listen (Arrays, Knotenlisten, Strings usw.)

Das Script fügt allen Arrays neue Methoden hinzu, die in Mozillas JavaScript-Spezifikation 1.6 definiert werden, aber noch nicht browserübergreifend umgesetzt werden. Mit diesen Methoden lassen sich alle möglichen Listen einfach durchlaufen, durchsuchen und Funktionen auf jedes Listenelement anwenden. Zudem stehen die Methoden für alle listenartige Objekttypen zur Verfügung, z.B. DOM-Knotenlisten und Strings.

Funktionen zum Ansprechen von Elementen nach ID und Klasse

Das Script definiert Funktionen mit den Kurznamen `$` bzw. `$$`. Mit der ersten lassen sich Elemente anhand ihrer ID, mit der zweiten anhand einer Klasse ansprechen. Mit langen Namen könnten Sie`getElementsById` und `getElementsByClassName` heißen.

Browserübergreifendes Event-Handling

Mithilfe der Funktionen `Event.add` und `Event.remove` können Sie komfortabel Event-Handler registrieren und entfernen. Die wichtigsten Browserunterschiede beim Event-Handling werden nivelliert.

...

...

...

forEach von Dean Edwards

http://dean.edwards.name/weblog/2006/07/enum/

ist eine Enumeration-Funktion für unterschiedliche Objekttypen

DOMhelp von Christian Heilmann

http://www.beginningjavascript.com/DOMhelp.js

ist ein Sammlung von Helferfunktionen aus Christian Heilmanns Buch »Beginning JavaScript with DOM Scripting and Ajax«.

Struktur zur Verkettung eigener Helferfunktionen von Dustin Diaz

http://www.dustindiaz.com/roll-out-your-own-interface/

Dustin Diaz zeigt, wie man sich eine eigenes kleines Framework im $(...)-Stil zusammenstellt.

ffjs von Sven Helmberger

http://fforw.de/ffjs/

minimalistische Bibliothek

Organisation von JavaScripten: Voraussetzungen und Überblick

Einleitung: Anforderungen an die heutige JavaScript-Programmierung

Dieser Artikel richtet sich an Web- und Anwendungsentwickler, die umfangreiche und komplexe JavaScripte schreiben. Mittlerweile ermöglichen Frameworks wie jQuery einen schnellen Einstieg in das DOM-Scripting. Wie gut der Code in diesen kleinen Anwendungsfällen strukturiert ist, spielt keine entscheidende Rolle. Das Framework gibt bereits eine Grundstruktur vor, die für diese Zwecke ausreicht.

Wie sehen wohlgestrukturierte umfangreiche JavaScript aus?

Wenn Scripte jedoch über ein simples DOM-Scripting hinausgehen und ganze Webanwendungen entstehen, stellen sich verschiedene Fragen: Wie lässt sich die komplexe Funktionalität geordnet in JavaScript implementieren? Welche Möglichkeiten der objektorientierten Programmierung in JavaScript gibt es? Wie lassen sich Datenmodelle in JavaScript angemessen umsetzen?

Eine sinnvolle Struktur wird zur Anforderung ebenso wie die Optimierung der Performance. Wird der JavaScript-Code mehrere tausend Zeilen lang, so soll er wartbar, übersichtlich und testbar bleiben. Er soll modular sein sowie wenig Redundanzen besitzen. Verschiedene Programmierer sollen gleichzeitig daran arbeiten können. Wenn das Script veröffentlicht werden soll, sodass es Fremde auf Ihren Sites verwenden, muss es in verschiedenen, unberechenbaren Umgebungen robust arbeiten.

Dieser Artikel soll Strategien vorstellen, um diesen Herausforderungen zu begegnen. Bevor wir uns konkreten JavaScript-Fragen widmen, soll zunächst erörtert werden, wann und wie JavaScript sinnvoll eingesetzt wird. Dies hat direkte Konsequenzen auf die Struktur und Arbeitsweise der JavaScripte. Deshalb ist es bereits auf dieser Ebene wichtig, die richtigen Weichen zu stellen. Alle weiteren Tipps gehen davon aus, dass Sie gemäß diesen Prinzipen arbeiten.

Das Schichtenmodell: Trennung von Inhalt, Präsentation und Verhalten

Im modernen Webdesign kommt den Webtechniken HTML, CSS und JavaScript jeweils eine bestimmte Rolle zu. HTML soll die Texte bedeutungsvoll strukturieren, indem z.B. Überschriften, Listen, Absätze, Datentabellen, Abschnitte, Hervorhebungen, Zitate usw. als solche ausgezeichnet werden. CSS definiert die Regeln für die Darstellung dieser Inhalte, sei es auf einem Desktop-Bildschirm, auf einem mobilen Gerät oder beim Ausdrucken.

Inhalte werden sinnvoll ausgez
Für die Struktur ist HTML zust
die Präsentation wird in Styles
ausgelagert.

Um eine Website effizient zu entwickeln sowie sie nachträglich mit geringem Aufwand pflegen zu können, sollen diese beiden Aufgaben strikt voneinander getrennt werden: Im HTML-Code werden keine Angaben zur Präsentation gemacht. Im Stylesheet befinden sich demnach alle Angaben zur Präsentation in möglichst kompakter Weise. Dadurch müssen im HTML-Code nur genau soviele Angriffspunkte für CSS-Selektoren gesetzt werden, wie gerade nötig sind (z.B. zusätzliche `div`- oder `span`-Elemente sowie`id`- und `class`-Attribute). Ein und dasselbe Dokument kann auf diese Weise durch den Wechsel des Stylesheets ein völlig anderes Layout bekommen. Aber auch ganz ohne Stylesheet sind die Inhalte noch sinnvoll strukturiert und die Inhalte zugänglich.

Erste Schritte zum Unobtrusive JavaScript

JavaScript kommt im Schichtenmodell die Aufgabe zu, dem Dokument**Verhalten** (engl. Behaviour) hinzuzufügen. Damit ist gemeint, dass das Dokument auf gewisse Anwenderereignisse reagiert und z.B. Änderungen im Dokument vornimmt.

Unaufdringliche JavaScripte si
getrennt vom HTML-Code. Sie
selbstständig, greifen auf das
Dokument zu und fügen Funkti
hinzu.

Die Grundlage für eine sinnvolle JavaScript-Programmierung ist die**Auslagerung des JavaScript-Codes**: Im HTML-Code sollte sich kein JavaScript in Form von Event-Handler-Attributen befinden (`onload`, `onclick`, `onmouseover` usw.). Stattdessen werden Elemente, denen ein bestimmtes Verhalten hinzugefügt werden soll, falls nötig mit einer Klasse oder ID markiert, um die Elemente eindeutung adressieren zu können. Die nötige Interaktivität wird dem Dokument automatisch hinzugefügt. Beim Ladens des Dokuments wird das Script aktiv, initialisiert sich und startet die Ereignisüberwachung an den betreffenden Elementen. Diese Anwendung von JavaScript nennt sich Unobtrusive JavaScript, »unaufdringliches« JavaScript.

JavaScript auslagern und extern am Dokumentende einbinden

Vermeiden Sie auf ins HTML eingebetteten JavaScript-Code. Verzichten Sie auf Inline-Event-Handler und `script`-Elemente, die direkt JavaScript enthalten. Binden Sie externe Scripte mit `<script type="text/javascript" src="..."></script>` ein. Aus Performance-Gründen sollten Sie dieses `script`-Element ans Dokumentende setzen, direkt vor den schließenden `</body>`-Tag.

Scripte bei DOM Ready initialisieren

Initialisieren Sie Ihre Script zu dem Zeitpunkt, wenn das HTML-Dokument eingelesen wurde und das DOM vollständig verfügbar ist. Dieser Zeitpunkt wird üblicherweise **DOM ready** genannt. SieheOnload-Techniken: Scripte ausführen, sobald das Dokument verfügbar ist. Die üblichen JavaScript-Frameworks stellen dafür Methoden bereit. Intern arbeiten diese hauptsächlich mit demDOMContentLoaded-Event. Wenn Sie kein Framework verwenden, gibt es auch lose Helferfunktionen für diesen Zweck, z.B. ContentLoaded von Diego Perini.

Beispiel jQuery:

```
$(document).ready(function () {
    // Diese Funktion wird beim DOM ready ausgeführt
});
```

Kurzschreibweise mit

```
$(function () {
    // ...
});
```

Übergabe einer Funktion über ihren Namen:

```
$(funktion);
```

DOM-Zugriffe über Selektor-Engines

Nutzen Sie JavaScript-Selektor-Engines, um auf das Dokument über CSS-artige Selektoren zuzugreifen und Elemente auszuwählen. Alle verbreiteten Frameworks bringen solche mit, sie sind aber auch separat erhältlich (z.B. Sizzle).

Beispiel jQuery:

```
$(document).ready(function () {
    // Liefert alle li-Elemente im Element mit der ID produktliste
    $('#produktliste li')
});
```

Zeitgemäßes Event-Handling

Nach den Ansprechen der Elemente können Sie Event-Handler registrieren. Dazu benötigen Sie eine leistungsfähige Event-Handling-Komponente, die sicherstellt, dass mehreren Handler für einen Typ registriert werden können und die Ihnen die Event-Verarbeitung durch Nivellierung von Browserunterschieden vereinfacht. Eine solche ist in den üblichen Frameworks eingebaut, alternativ können Sie eine lose addEvent-Helferfunktionen nutzen.

Beispiel jQuery:

```
$(document).ready(function () {
    $('#produktliste li').click(function (e) {
        // ... Verarbeite Ereignis ...
    });
});
```

Zum effizienten Event-Handling gehört Event Delegation hinzu. Die Grundidee ist, dass Ereignisse bei einer ganzen Schar von Elementen durch ein darüberliegendes Element verarbeitet werden. Dazu macht man sich das Event Bubbling und Event Capturing zunutze. Beispielsweise jQuery bietet dafür die Methoden delegate() und live().

Objektorientierte Programmierung in JavaScript

JavaScript zieht viele Programmieranfänger an und ist für viele Webautoren die erste Programmiersprache, mit der sie aktiv in Kontakt kommen. Für sie bleibt die genaue Funktionsweise von JavaScript zunächst unzugänglich. Sie treffen auf Fallstricke und schaffen es nicht, JavaScript zu bändigen. Aber auch professionelle Software-Entwickler mit fundierten Kenntnissen anderer Programmiersprachen geraten an JavaScript. Sie sind nicht weniger verwirrt und verzweifeln, weil sie in JavaScript nicht Strukturen wiederfinden, die ihnen in anderen Sprachen Orientierung bieten.

Klassenbasierte Objektorientierung in anderen Programmiersprachen

Viele verbreitete Programmiersprachen arbeiten zumindest teilweise objektorientiert und besitzen ausgereifte, sehr konventionelle Konzepte zur Strukturierung von Programmen. Das sind vor allem **Klassen** und ferner **Module**,**Pakete** bzw. **Namensräume**.

OOP in vielen anderen Programmiersprachen läuft üb
Klassen, die Kapselung und vie
erlauben. Zusammenhängende
werden in Namensräumen bzv
Modulen gruppiert.

Klassen werden üblicherweise mittels einer Deklaration beschrieben und können daraufhin verwendet werden. Sie bestehen aus Konstruktoren, Methoden und Eigenschaften. Sie können von anderen Klassen erben und diese erweitern. Von Klassen lassen sich beliebig viele Instanzen anlegen – oder nur eine Instanz im Falle von Singletons. Die Sichtbarkeit bzw. Verfügbarkeit von Methoden und Eigenschaften kann über Schlüsselwörter geregelt werden. So wird z.B. zwischen öffentlichen und privaten Eigenschaften und Methoden unterschieden. Eigenschaften können als nur lesbare Konstanten definiert werden. Darüber hinaus erlauben einige Programmiersprachen die Definition von Interfaces, ein Anforderungskatalog mit Methoden, die eine Klasse bereitstellen muss. Verbreitet sind ferner statische Methoden (Klassenmethoden) sowie Getter- und Setter-Methoden, die beim Lesen bzw. Schreiben von Eigenschaften aufgerufen werden.

Diese konventionelle klassenbasierte OOP sieht beispielsweise in PHP 5.3 folgendermaßen aus:

```
namespace Beispielnamensraum;
```

```
interface IBeispielInterface
{
    public function oeffentlicheMethode();
}

class Beispielklasse implements IBeispielInterface
{
    function __construct() {
        echo "Konstruktor\n";
    }

    public $oeffentlich = 'Öffentliche Eigenschaft';
    private $privat = 'Private Eigenschaft';

    public function oeffentlicheMethode() {
        echo "Öffentliche Methode\n";
        $this->privateMethode();
    }
    private function privateMethode() {
        echo "Private Methode\n";
        echo $this->privat . "\n";
    }

    const konstante = 'Klassenkonstante';
    public static $statisch = 'Statische Eigenschaft (Klasseneigenschaft)';
    public static function statischeMethode() {
        echo "Statische Methode (Klassenmethode)\n";
        echo self::$statisch . "\n";
        echo self::konstante . "\n";
    }
}

class AbgeleiteteKlasse extends Beispielklasse
{
    function __construct() {
        parent::__construct();
        print "Konstruktor von AbgeleiteteKlasse\n";
    }

    public function zusatzmethode() {
        echo "Zusatzmethode von AbgeleiteteKlasse\n";
        parent::oeffentlicheMethode();
    }
}

$instanz = new Beispielklasse();
$instanz->oeffentlicheMethode();
echo $instanz->oeffentlich . "\n";

echo Beispielklasse::statischeMethode();

$instanz2 = new AbgeleiteteKlasse();
$instanz2->zusatzmethode();
```

Dieses Beispiel soll hier nicht näher beschrieben werden, sondern nur den Hintergrund illustrieren für diejenigen, die solche oder ähnliche klassenbasierte OOP bereits kennen.

Die besagten Konzepte bilden in eine Richtschnur: Hält sich ein Programmierer an diese Konventionen, so ist mehr oder weniger garantiert, dass das Programm von anderen grob verstanden und wiederverwendet werden kann und nicht mit anderen Programmen in

die Quere kommt. Selbst Sprachen, die nicht konsequent objektorientiert arbeiten, bekommen eine einheitliche Struktur dadurch, dass die Programmierer ihren Code in Namensräumen und Klassen organisieren.

JavaScript hat keine Klassen – doch das ist nicht schlimm

ECMAScript 3, der JavaScript zugrunde liegende Webstandard, besitzt all diese Features nicht. Und der Nachfolger ECMAScript 5 verbessert lediglich die eigentümlichen Fähigkeiten, bietet etwa Methoden zur Datenkapselung, fügt jedoch absichtlich keine Klassen in JavaScript ein.

JavaScript bietet keine Patentlö wie Klassen, sondern erlaubt verschiedene Programmier-paradigmen. Man muss selbst geeignete Techniken zur Strukturierung wählen.

JavaScript bringt von Haus aus keine »einzig wahre« Programmiertechnik wie Namensräume und Klassen mit. Das kann man als Nachteil auffassen, aber auch als Vorteil. Weil JavaScript ein vorgegebenes Gerüst zu fehlen scheint, fehlt Einsteigern die Orientierung. Es gibt nicht den einen vordefinierten Weg, ein Programme in JavaScript zu strukturieren. Stattdessen muss man sich selbst über die sinnvolle Organisation von JavaScripten Gedanken machen.

Glücklicherweise ist JavaScript so leistungsfähig, dass sich die besagten Strukturen durchaus mit JavaScript umsetzen lassen. Es ist mit Zusatzbibliotheken möglich, JavaScript klassenbasiert zu entwickeln. Dies vereinfacht den Einstieg für diejenigen, die bereits andere, klassenbasierte Programmiersprachen beherrschen.

Dabei sollten allerdings nicht die Eigenarten und besonderen Fähigkeiten von JavaScript vergessen werden. Im Gegensatz zu manch anderen klassenbasierten Sprachen ist JavaScript äußerst dynamisch und besitzt funktionale Aspekte. JavaScript-Kenner empfehlen, die Fähigkeiten von JavaScript zu nutzen, anstatt bloß klassenbasierte OOP überzustülpen. Mit diesen Möglichkeiten ist JavaScript nicht unbedingt schlechter und defizitär im Vergleich zu klassenbasierter Objektorientierung – es ist lediglich ein anderer, nicht minder interessanter und brauchbarer Ansatz.

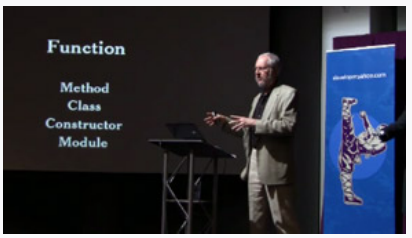
Grundpfeiler der fortgeschrittenen JavaScript-Programmierung

Die Grundlage von ECMAScript 3 sind Objekte, die zur Laufzeit beliebig um Eigenschaften und Methoden ergänzt werden können. Ein JavaScript-Objekt ist eine ungeordnete Liste, in der String-Schlüsseln beliebige Werte zugeordnet werden. Diese dynamischen, erweiterbaren Objekte ermöglicht eine Strukturierung von Programmen. Sie dienen, wie wir später sehen werden, als vielseitiges Mittel zur Gruppierung und sind insofern mit Hashes, Namensräumen, Singletons bzw. Klassen mit statischen Methoden in anderen Sprachen vergleichbar.

JavaScript ist eine objektorient dynamische Sprache. Funktion Prototypen machen sie enorm leistungsfähig.

Die wirklichen Stärken von JavaScript liegen jedoch woanders: **Funktionen**. Dazu Douglas Crockford in dem Vortrag Crockford on JavaScript – Act III: Function the Ultimate:

> Functions are the very best part of JavaScript. It's where most of the power is, it's where the beauty is. … Function is the key idea in JavaScript. It's what makes it so good and so powerful. In other languages you've got lots of things: you've got methods, classes, constructors, modules, and more. In JavaScript there's just function, and function does all of those things and more. That's not a deficiency, that's actually a wonderful thing — having one thing that can do a lot, and can do it brilliantly, at scale, that's what functions do in this language.



Crockford weist hier darauf hin, wie vielseitig Funktionen in JavaScript sind und dass sie Aufgaben übernehmen, für die es in anderen Sprachen unterschiedliche Konzepte gibt.

Das folgende Diagramm soll die Säulen der fortgeschrittenen JavaScript-Programmierung illustrieren. Das Fundament bilden dynamische Objekte und `Object`-Literale, darauf bauen Funktionen auf, welche Scopes erzeugen und als Konstruktoren prototypische Vererbung ermöglichen.

Objekte / `Object`-Objekte

- Alles ist ein Objekt – bis auf Primitives, die sich allerdings wie Objekte verhalten
- Objekte sind i.d.R. erweiterbar (ECMAScript 3)
- Object-Objekte sind Allround-Container und Hashes
- Object-Literale `{ key : value, ... }`

Funktionen

- Dienen der Codestrukturierung
- Objekte erster Klasse: Zur Laufzeit erzeugen, übergeben, zurückgeben, in Variablen speichern
- Funktionale Programmierung (z.B. Listenoperationen, Currying, Event-Handler)
- Können an Objekte als Methoden gehängt werden
- Methoden haben `this`-Kontext, über `call` und `apply`veränderbar (Binding)
- Erzeugen Scopes
- Können verschachtelt werden (Scope-Chain)
- Dienen als Konstruktoren
- Besitzen Prototypen für damit erzeugte Objekte (`funktion.prototype`)

Scope

- Gültigkeitsbereich für Variablen
- Allzweckwerkzeug für Datenverfügbarkeit und Kapselung/Privatheit
- Nur Funktionen erzeugen einen Scope, keine anderen Blöcke
- Identifier Resolution über die Scope-Chain (`bezeichner` wird über Scope-Chain aufgelöst)
- Scope-Chain speichert die internen Variablenobjekte von Funktionsaufrufen
- Scope-Chain ermöglicht Closures und darüber Currying

Prototypen

- Prototypenbasierte Vererbung
- `objekt.eigenschaft` wird über die Prototype-Chain aufgelöst
- Ganz normale Objekte, die im Programm und nicht per Deklaration erzeugt werden
- Ein Objekt stellt Funktionalität für ein anderes bereit (Delegation)
- Erzeugen von Objekten gleicher Funktionalität unter Schonung des Speichers
- Ableiten und Redefinieren von Funktionalität
- Keine Trennung zwischen Deklaration und Instanz – alle Objekte können Prototypen sein

Dieses Diagramm soll Ihnen einen kompakten Überblick geben. Viele der Begriffe soll dieser Artikel im weiteren erläutern, auf andere kann er nicht eingehen, denn dazu ist eine systematische Einführung in JavaScript nötig.

Organisation von JavaScripten: Module und Kapselung

Unstrukturierte Scripte

Zahlreiche Scripte, die JavaScript-Programmierer im Netz anbieten, liegen in einer gesonderten Datei vor und sind darüber hinaus unstrukturiert. Es handelt sich um eine lose Sammlung von dutzenden globalen Variablen und Funktionen:

```
var variable1 = "wert";
var variable2 = "wert";
var variable3 = "wert";

function funktion1 () {
    /* ... */
```

```
}
function funktion2 () {
    /* ... */
}
function funktion3 () {
    /* ... */
}
```

Diese Organisation bringt in der Regel mit sich, dass das Script nicht einfach konfigurierbar, anpassbar und erweiterbar ist. Am schwersten wiegt jedoch, dass es sich um eine große Zahl von losen Objekten im globalen Scope (Variablen-Gültigkeitsbereich) handelt. Globale Variablen und Funktionen sind Eigenschaften des `window`-Objektes. Das obige Beispiel definiert daher sechs Eigenschaften beim`window`-
Objekt: `window.variable1` bis `indow.variable3` sowie `window.funktion1` bis`window.funktion3`.

Das Yin und Yang von JavaScript: Kapselung vs. Verfügbarkeit

Clientseitige JavaScripte arbeiten unter besonderen Bedingungen: Ein Script operiert auf einem HTML-Dokument, auf das es über das DOM zugreift. Ferner operiert es im Kontext des sogenannten globalen Objektes. Das ist in JavaScript das `window`-Objektes, welches den obersten Namensraum bereitstellt. Sowohl das globale Objekt als auch das DOM teilt es sich mit anderen Scripten. Diese »öffentlichen Güter« darf kein Script für sich alleine beanspruchen.

Unstrukturierte Scripte sind sc
zu warten und kollidieren mit
Scripten. Vermeiden Sie global
Variablen, soweit möglich.

Wenn Scripte unterschiedlicher Herkunft zusammenkommen, kann das schnell zu Konflikten führen. Die Vermeidung von Konflikten setzt bereits beim Unobtrusive JavaScript an: Indem wir JavaScript-Code nicht direkt ins HTML-Dokument einbetten, sondern fortgeschrittenes Event-Handling verwenden, reduzieren wir Überschneidungen im DOM.

Konfliktfeld Nummer Eins bleibt das `window`-Objekt. Darüber können über das zwei Scripte zusammenarbeiten, aber auch in Konflikt geraten, wenn sie gleichnamige Variablen definieren. In JavaScript gilt es daher, ein **Gleichgewicht zwischen Kapselung und Verfügbarkeit** herzustellen.

Datenkapselung bedeutet, dass das Erweitern des globalen Objekts sowie der DOM-Objekte auf ein Minimum reduziert wird. Ein Script sollte den globalen Scope nicht für seine Arbeitsdaten verwenden und globale Variablen möglichst vermeiden. Es sollte nur die Objekte am `window`-Objekt speichern, die für den Zugriff von außen unbedingt vonnöten sind.

Bei manchen Aufgaben ist es möglich, ein Script konsequent zu kapseln, sodass es das globale `window`-Objekt nicht antastet. In anderen Fällen ist es nötig, zumindest einige Objekte global verfügbar zu machen. Gründe dafür können sein:

Die öffentliche API ihres Script
benötigt nur ein globales Objek
welches die restlichen Funktio
zugänglich sind.

- Eine öffentliche Programmierschnittstelle (API) besonders bei Bibliotheken
- Konfigurierbarkeit des Scripts z.B. durch verschiedene Nutzer
- Erweiterbarkeit (Modularisierung)

Es kommt daher auf das richtige Gleichgewicht an. Zwei Beispiele: Das riesige jQuery-Framework definiert standardmäßig nur zwei globale Variablen: `window.jQuery()` und als Alias `window.$()`. Das YUI-Framework definiert lediglich `window.YUI()`. `window.jQuery()` und `window.YUI()` sind beides Funktionen, denen man beim Aufruf letztlich Funktionen übergibt - dazu später mehr. Beide Frameworks schaffen es, nicht mehr als ein globales Objekt anzulegen, ohne auf die obigen Features wie Erweiterbarkeit zu verzichten.

Einfache Module mit dem Objekt-Literal

Eine einfache Möglichkeit, um den globalen Scope zu schonen, ist die Gruppierung aller Variablen und Funktionen eines Scripts in einer JavaScript-Objektstruktur. Im globalen Geltungsbereich taucht dann nur noch diese eine Objektstruktur auf, andere globale Variablen oder Funktionen werden nicht belegt. Das Script ist in der Objektstruktur in sich abgeschlossen. Damit sind Wechselwirkungen mit anderen Scripten ausgeschlossen, solange der Bezeichner der Objektstruktur eindeutig ist.

`Object`-Objekte sind besonder
vielseitig. Sie sind das Grundw
zur Gruppierung von Objekten
damit zur Strukturierung von
Programmen. Sie sind als Hash
allgegenwärtig.

Ein JavaScript-Objekt ist erst einmal nichts anderes als ein Container für weitere Daten. Ein Objekt ist eine Liste, in der unter einem Bezeichner gewisse Unterobjekte gespeichert sind. Aus anderen Programmiersprachen

ist diese solche Datenstruktur als *Hash* oder *assoziativer Array* bekannt. In JavaScript sind alle vorgegebenen Objekte und Methoden in solchen verschachtelten Objektstrukturen organisiert, z.B. `window.document.body`.

In JavaScript gibt es den allgemeinen Objekttyp `Object`, von dessen Prototypen alle anderen JavaScript-Objekte abstammen. Das heißt, jedes JavaScript-Objekt ist immer auch ein `Object`-Objekt.`Object` ist die Grundlage, auf der die restlichen spezifischeren Objekttypen aufbauen.

Für die Organisation von eigenen Scripten bieten sich solche unspezifischen `Object`-Objekte an. Über `new Object()` lässt sich ein `Object`-Objekt erzeugen:

```
var Modul = new Object();
Modul.eigenschaft = "wert";
Modul.methode = function () {
    alert("Modul-Eigenschaft: " + Modul.eigenschaft);
};
Modul.methode();
```

Über die gewohnte Schreibweise zum Ansprechen von Unterobjekten (`objekt.unterobjekt`) werden dem `Object` weitere Objekte angehängt. Im Beispiel werden zwei Objekte angehängt, ein String und eine Funktion.

Der Name `Modul` ist selbstverständlich nur als Platzhalter gemeint. Sie sollten das `Object`-Objekt (im Folgenden kurz `Object` genannt) eindeutig und wiedererkennbar nach der Aufgabe bzw. dem Zweck ihres Scriptes benennen.

JavaScript bietet für das Definieren von `Object`-Objekten eine Kurzschreibweise an, den sogenannten **Object-Literal**. Ein `Object`-Literal beginnt mit einer öffnenden geschweiften Klammer `{` und endet mit einer schließenden geschweiften Klammer `}`. Dazwischen befinden sich, durch Kommas getrennt, die Zuweisungen von Namen zu Objekten. Zwischen Name und Objekt wird ein Doppelpunkt notiert. Das Schema ist also: `{ name1 : objekt1, name2 : objekt2, … nameN : objektN }`

> Der `Object`-Literal erlaubt da
> kompakte Erzeugen von `Objec`
> Objekten und eignet sich hervo
> für die Definition von Modulen.

Das obige Beispiel-`Object` lässt sich in der Literalschreibweise so umsetzen:

```
var Modul = {
    eigenschaft : "wert",
    methode : function () {
        alert("Modul-Eigenschaft (über window.Modul): " + Modul.eigenschaft);
        // Alternativ:
        alert("Modul-Eigenschaft (über this): " + this.eigenschaft);
    }
};
Modul.methode();
```

Eine Illustration der entstehenden Verschachtelung:

- `window` (globales Objekt)
    - `Modul` (Object)
        - `eigenschaft` (String)
        - `methode` (Function)

Der Zugriff auf die Unterobjekte des `Object`-Containers ist von außen über den globale Namen nach dem Schema `Modul.eigenschaft` möglich. Im Beispiel wird über `Modul.methode()` die zuvor angehängte Funktion aufgerufen.

Kapselung mit privatem Funktions-Scope

Beim Objekt-Literal wird ein globales Objekt als Namensraum benutzt, um darin eigene Objekte unterzubringen. All diese

Objekte sind über das Containerobjekt für andere Scripte zugänglich. Es gibt also keine Trennung zwischen öffentlichen und privaten Daten. Während es sinnvoll ist, dass z.B. eine Methode `Modul.methode()` von außen aufrufbar ist, ist es unnötig und potenziell problematisch, dass jede Objekteigenschaft gelesen und manipuliert werden kann.

Wirksame Kapselung erreiche
einer Funktion, die Ihre Variab
einschließt und nur wenige Ob
nach außen verfügbar macht.

Der nächste Schritt ist daher, eine wirksame Kapselung zu implementieren. Das Mittel dazu ist ein eigener, privater Scope (Variablen-Gültigkeitsbereich). Darin können beliebig viele lokale Variablen und Methoden definiert werden. Die einzige Möglichkeit, in JavaScript einen Scope zu erzeugen, ist eine Funktion. Wir definieren also eine Funktion, um darin das gesamte Script zu kapseln. Solange durchgehend lokale Variablen und Funktionen verwendet werden, wird der globale Scope nicht angetastet.

Ein mittlerweile stark verbreitetes Muster ist daher folgender Codeschnipsel:

```
(function () {
    /* ... */
})();
```

Schließen Sie Ihren Code in ein
Funktionsausdruck ein, der sof
ausgeführt wird. Darin können
Objekten quasen, ohne den glo
Scope zu verpesten.

Dies erscheint zunächst sehr kryptisch, daher eine schrittweise Zerlegung der Syntax:

1. Erzeuge eine namenlose Funktion per Funktionsausdruck: `function () { ... }`
2. Umschließe diesen Funktionsausdruck mit runden Klammern: `(function () {})`
3. Führe die Funktion sofort aus mit dem Call-Operator, das sind die beiden runden Klammern:`(function () { ... })()`. Die Parameterliste bleibt in diesem Beispiel leer.
4. Schließe die Anweisung mit einem `;` ab.

Diese anonyme Funktion wird nur notiert, um einen Scope zu erzeugen, und sie wird sofort ausgeführt, ohne dass sie irgendwo gespeichert wird. Innerhalb der Funktion wird nun der gewünschte Code untergebracht:

```
(function () {

    /* Lokale Variable */
    var variable = 123;

    /* Lokale Funktion */
    function funktion () {
        /* ... */
    }

    /* Rufe lokale Funktion auf: */
    funktion();

    /* Zugriff auf globale Objekte ist ebenfalls möglich: */
    alert(document.title);

})();
```

Im Beispiel finden sich eine Variablendeklarationen und eine Funktionsdeklaration. Beide sind lokal, sind also nur innerhalb der Kapselfunktion zugänglich. Wir können auf die Variablen und Funktionen direkt zugreifen.

Das Beispiel macht noch nichts sinnvolles. Die Nützlichkeit von Funktionen zur Kapselung ergibt sich z.B. bei einem Anwendungsbeispiel mit Event-Handling.

Vergessen Sie nicht, Variablen
mit `var`als lokal zu deklarieren
Andernfalls werden sie automa
global, also Eigenschaften von

```
(function () {

    var clickNumber = 0;
    var outputEl;
```

```
    function buttonClicked () {
        clickNumber++;
        outputEl.html('Button wurde ' + clickNumber + ' Mal angeklickt');
    }

    function init () {
        outputEl = jQuery('#output);
        jQuery('#button').click(buttonClicked);
    }

    jQuery(document).ready(init);

})();
```

Das zugehörige HTML:

```
<button id="button">Klick mich</button>
<p id="output">Button wurde noch nicht angeklickt</p>
<script type="text/javascript" src="beispiel.js"></script>
```

Der Code nutzt die jQuery-Bibliothek, um eine Initialisierungsfunktion bei DOM ready auszuführen. Diese registriert bei einem Button einen Event-Handler. Wird der Button geklickt, wird eine Zahl erhöht. Zudem wird die bisherige Anzahl der Klicks im Dokument ausgegeben.

Das Besondere an diesem Script sind die vier lokalen Variablen bzw. Funktionen. Sie werden direkt im Funktions-Scope notiert, anstatt sie an einen `Object`-Container zu hängen. Innerhalb der verschachtelten Funktionen sind die Variablen des äußeren Funktions-Scope verfügbar (siehe Closures). `init()` füllt die Variable `outputEl` und greift auf die Funktion `buttonClicked()` zu. `buttonClicked()` greift auf die Variablen `clickNumber` und `outputEl` zu. Das Script funktioniert, ohne dass Objekte am globalen `window`-Objekt angelegt werden.

DOM-ready-Handler als privater Scope

Bei der Verwendung mit jQuery ist das Anlegen solcher Funktions-Scope gang und gäbe. Wenn die Initialisierung eines Scriptes auf DOM ready warten soll, dann übergibt man einen Funktionausdruck an `jQuery(document).ready()`. Diese Funktion wird als Handler beim Eintreten des DOM-ready-Ereignisses ausgeführt. Man nutzt sie gleichzeitig als privaten Scope für weitere Objekte. Das obige Beispiel können wir also folgendermaßen anpassen:

DOM-Ready-Handlerfunktione
verschiedenen Bibliotheken bi
berets einen privaten Scope, d
nutzen sollten.

```
jQuery(function ($) {

    var clickNumber = 0;
    var outputEl;

    function buttonClicked () {
        clickNumber++;
        outputEl.html('Button wurde ' + clickNumber + ' Mal angeklickt');
    }

    function init () {
        outputEl = $('#output);
        $('#button').click(buttonClicked);
    }

    init();

});
```

Die übergebene DOM-ready-Funktion bekommt das globale jQuery-Objekt als ersten Parameter. Wir nennen den Parameter

hier `$`. Funktionsparameter sind automatisch lokale Variablen, das heißt, wir können mit `$` genauso umgehen wie mit `clickNumber` oder `buttonClicked`.

Globale Objekte importieren

jQuery stellt standardmäßig `window.$` als Abkürzung für `window.jQuery` zur Verfügung, wenn nicht der noConflict-Modus aktiviert wird. Es ergibt jedoch Sinn, das jQuery-Objekt als lokale Variable zu definieren, denn das beschleunigt den Zugriff darauf (Stichwort Scope-Chain).

Das Übergeben von Objekten in Kapselfunktion verkürzt die Sc Kette und beschleunigt den Zu diese Objekte etwas.

Aus demselben Grund hat es sich eingebürgert, das `window`-Objekt sowie weitere häufig benutzte Objekte wie `document` mittels Parametern in den Funktions-Scope zu übergeben:

```
(function (window, document, undefined) {

    /* ... */

})(window, document);
```

Gleichzeitig wird hier sichergestellt, dass innerhalb der Funktion der Bezeichner `undefined` immer den Typ `Undefined` besitzt. Wir definieren einen solchen Parameter, aber übergeben keinen Wert dafür – sodass eine lokale Variable namens `undefined` mit einem leeren Wert angelegt wird. Das ist andernfalls nicht garantiert, denn `window.undefined` ist durch Scripte überschreibbar.

Innerhalb der Funktion können die Objekte genauso heißen wie außerhalb. Dennoch handelt es z.B. bei `document` innerhalb der Funktion um eine lokale Variable, auch wenn sie natürlich auf `window.document` verweist.

Das Revealing Module Pattern: Kapselung plus öffentliche Schnittstelle

Wir haben nun beide Extreme kennengelernt: Bei `Object`-Containern sind alle Unterobjekte öffentlich. Bei einer Kapselfunktion ist kein Objekt nach außen hin zugänglich. Wenn wir ein wiederverwendbares Script schreiben wollen, wollen wir meist eine öffentliche Schnittstelle (API) anbieten. Dazu müssen einige ausgewählte Objekte, in der Regel Methoden, sowohl nach außen sichtbar sein als auch Zugriff auf die internen, privaten Objekte haben. Man spricht in diesem Fall von **privilegierten Methoden**.

Diesen Kompromiss erreichen wir durch eine Kombination aus `Object`-Literalen und einer Kapselfunktion. Dieses Entwurfsmuster nennt sich **Revealing Module Pattern**. Kurz gesagt gibt die Kapselfunktion ein Objekt nach draußen, bevor sie sich beendet. Über dieses Objekt können gewisse privilegierte Methoden aufgerufen werden.

Das Revealing Module Pattern öffentliche und private Objekte eignet sich ideal, um API und i Implementierung sauber zu tr

Wir beginnen mit dem bereits beschriebenen Funktionsausdruck, der sofort ausgeführt wird:

```
(function () {
    /* ... private Objekte ... */
})();
```

Das Neue ist, dass diese Funktion einen Wert zurückgibt, der in einer Variable gespeichert wird:

```
var Modul = (function () {
    /* ... private Objekte ... */
})();
```

Dieser Wert ist ein Objekt, welches wir in der Funktion mit einem Objekt-Literal notieren und mittels `return` nach draußen geben. An dem Objekt hängen die öffentlichen Eigenschaften und Methoden:

```
var Modul = (function () {

    /* ... private Objekte ... */
```

```
    /* Gebe öffentliche API zurück: */
    return {
        öffentlicheMethode : function () { ... }
    };

})();
```

Innerhalb der anonymen Funktion notieren wir wie üblich unsere privaten Objekte. Das folgende Beispiel definiert eine öffentliche, privilegierte Methode. Sie hat Zugriff auf sämtliche internen, privaten Objekte, welche direkt von außen nicht zugänglich sind.

```
var Modul = (function () {

    // Private Objekte
    var privateVariable = "privat";
    function privateFunktion () {
        alert("privateFunktion wurde aufgerufen\n" +
            "Private Variable: " + privateVariable);
    }

    // Gebe öffentliches Schnittstellen-Objekt zurück
    return {
        öffentlicheMethode : function () {
            alert("öffentlicheMethode wurde aufgerufen\n" +
                "Private Variable: " + privateVariable);
            privateFunktion();
        }
    };

})();

// Rufe öffentliche Methode auf
Modul.öffentlicheMethode();

// Ergibt undefined, weil von außen nicht sichtbar:
window.alert("Modul.privateFunktion von außerhalb: " + Modul.privateFunktion);
```

Da die privilegierten Methoden innerhalb des Funktions-Scope notiert werden, haben sie darauf Zugriff. Das liegt daran, dass sie Closures sind.

Es ist natürlich möglich, solche Module nicht direkt als globale Variablen zu speichern, sondern verschiedene in einem `Object`-Literal zu speichern. Dieser dient dann als Namensraum für zusammengehörige Module. So ist letztlich mehrere Module unter nur einer globalen Variable gespeichert.

Module können Sie mit einem `Object`in einem Namen gruppieren.

```
var Namensraum = {};
Namensraum.Modul1 = (function () { ... })();
Namensraum.Modul2 = (function () { ... })();
```

Erweiterbare Module

Ben Cherry schlägt eine Erweiterbarkeit von Modulen auf Basis des Revealing Module Patterns vor. Er unterscheidet zwischen fester und lockerer Kopplung der Teile. Das heißt, entweder setzt ein Aufbaumodul ein Basismodul zwingend voraus. Oder beide Module ergänzen sich gegenseitig, sind aber auch separat funktionsfähig.

Module nachträglich zu erweit möglich, allerdings haben die e Teile keinen Zugriff auf die priv Objekte der anderen Teilmodu

Feste Kopplung

```
/* Grundmodul */
var Modul = (function (Modul) {
    /* ... private Objekte ... */
    return {
        methode1 : function () { ... }
    };
})();

/* Erweiterung des Grundmoduls */
(function (modul) {
    /* ... private Objekte ... */
    /* Erweitere Modul um neue Methoden: */
    modul.methode2 = function () { ... };
})(Modul);
```

Die Definition des Grundmoduls erfolgt wie beim Revealing Module Pattern besprochen. Zur Erweiterung des Moduls wird eine weitere anonyme Funktion angelegt und ausgeführt. Diese Funktion bekommt das Modulobjekt als Parameter übergeben und fügt diesem neue Methoden hinzu oder überschreibt vorhandene. Innerhalb der Funktion können wie üblich private Objekte und Methoden angelegt werden.

Nach der Ausführung des obigen Codes besitzt das Modul zwei öffentliche Methoden:

```
Modul.methode1();
Modul.methode2();
```

Zu beachten ist, dass die Methoden der Erweiterung keinen Zugriff auf die privaten Objekte des Grundmoduls haben – denn sie befinden sich in einem anderen Funktions-Scope. Zur Lösung dieses Problems schlägt Ben Cherry eine Methode vor, die die privaten Objekte kurzzeitig öffentlich macht, sodass ein übergreifender Zugriff möglich ist. Das erscheint mir jedoch besonders umständlich – in diesem Fall würde ich privaten Objekte zu dauerhaft öffentlichen Eigenschaften machen und auf die vollständige Kapselung verzichten.

Lose Kopplung

Bei der losen Kopplung können die Teilmodule alleine oder zusammen stehen. Ferner ist die Reihenfolge, in der die Teilmodule notiert werden, unwichtig. Dafür können sie nicht stillschweigend auf die gegenseitigen öffentlichen Methoden zugreifen, sondern müssen gegebenenfalls prüfen, ob diese definiert sind.

```
var Modul = (function (modul) {
    /* ... private Objekte ... */

    /* Lege Methode am Modulobjekt an: */
    modul.methode1 = function () { ... };

    return modul;
}(Modul || {}));

var Modul = (function (modul) {
    /* ... private Objekte ... */

    /* Lege Methode am Modulobjekt an: */
    modul.methode2 = function () { ... };

    return modul;
}(Modul || {}));
```

Die Moduldeklarationen sind gleich aufgebaut: Es gibt eine anonyme Funktion, um einen privaten Scope zu erzeugen. Diese Funktion bekommt das bestehende Modul übergeben. Der Ausdruck `Modul || {}` prüft, ob das Modul bereits definiert wurde. Falls ja, wird dieses der Funktion übergeben. Andernfalls wird mit dem `Object`-Literal ein leeres Objekt erzeugt und übergeben. Somit ist gesichert, dass die Funktion ein Objekt als Parameter entgegennimmt. Innerhalb der Funktion können wir private Objekte notieren und

das Modulobjekt um neue Eigenschaften erweitern. Am Ende wird das Modul zurückgegeben und der Rückgabewert in einer Variable gespeichert.

Das Resultat ist ebenfalls, dass das Modul zwei öffentliche Methoden besitzt:

```
Modul.methode1();
Modul.methode2();
```

Organisation von JavaScripten: Konstruktoren, Prototypen und Instanzen

Objektinstanzen mit Konstruktoren und Prototypen

Mit `Object`-Literalen und dem Revealing Module Pattern haben wir das gebaut, was in anderen Programmiersprachen *Singleton* oder *Klasse* mit statischen Methoden genannt wird. Ein solches Modul kommt nur einmal vor und kann nur einen internen Status haben. In vielen Fällen ist es jedoch sinnvoll, mehrere Instanzen (Exemplare) eines Objektes zu erzeugen. In vielen anderen Programmiersprachen würde man dazu eine eigene Klasse definieren und davon Instanzen erzeugen.

In JavaScript gibt es wie eingangs erwähnt keine Klassen. Es gibt jedoch **Konstruktor-Funktionen**(kurz: **Konstruktoren**) und **prototypische Objekte** (kurz: Prototypen).

Konstruktor-Funktionen (Konstruktoren)

Der Name Konstruktor stammt vom englischen *construct* = erzeugen, konstruieren, bauen. Eine Konstruktor-Funktion ist demnach ein Erzeuger neuer Objekte. Sie werden sich sicher fragen, wie die Syntax zum Notieren von Konstruktoren lautet. Ein Konstruktur ist jedoch keine besondere Sprachstruktur, sondern erst einmal eine ganz normale Funktion. Zu einem Konstruktor wird sie lediglich dadurch, dass sie mit dem Schlüsselwort `new` aufgerufen wird.

> Konstruktoren sind normale Funktionen, die mit `new Funktion`aufgerufen werden. erzeugte Instanzobjekt ist dari über `this`verfügbar.

Wenn eine Funktion mit `new` aufgerufen wird, wird intern ein neues, leeres `Object`-Objekt angelegt und die Funktion im Kontext dieses Objektes ausgeführt. Das bedeutet, im Konstruktor kann das neue Objekt über `this` angesprochen werden. Darüber können ihm z.B. Eigenschaften und Methoden hinzugefügt werden.

Intern wird also ein `Object`-Objekt angelegt, genauso wie wir es zur Strukturierung getan haben. Der Unterschied beim Konstruktor ist, dass auf diese Weise unzählige gleich ausgestattete Objekte, sogenannte **Instanzen** erzeugt werden können.

```
// Konstruktorfunktion
function Konstruktor () {
    // Zugriff auf das neue Objekt über this,
    // Hinzufügen der Eigenschaften und Methoden
    this.eigenschaft = "wert";
    this.methode = function () {
```

```
        // In den Methoden wird ebenfalls über this auf das Objekt zugegriffen
        alert("methode wurde aufgerufen\n" +
              "Instanz-Eigenschaft: " + this.eigenschaft);
    };
}

// Erzeuge Instanzen
var instanz1 = new Konstruktor();
instanz1.methode();
var instanz2 = new Konstruktor();
instanz2.methode();
// usw.
```

Dieses Beispiel enthält eine Konstruktorfunktion, mithilfe derer zwei Instanzen erzeugt werden. Innerhalb des Konstruktors werden dem leeren Instanzobjekt zwei Eigenschaften hinzugefügt, ein String und eine Funktion.

Als Funktion kann ein Konstruktor Parameter entgegennehmen. Somit können Instanzen mit unterschiedlichen Eigenschaften erzeugt werden:

```
function Katze (name, rasse) {
    this.name = name;
    this.rasse = rasse;
    this.pfoten = 4;
}
var maunzi = new Katze('Maunzi', 'Perserkatze');
```

Die dem Konstruktor übergebenen Parameter werden zu Eigenschaften des Instanzobjekts. Auch wenn verschiedene `Katze`-Instanzen abweichende Eigenschaftswerte haben können, so ist ihnen allen gemein, dass sie die Eigenschaften `name` und `rasse` besitzen. Neben diesen beiden gibt es eine feste Eigenschaft `pfoten`: Katzen haben (in der Regel) vier Pfoten.

Eigenschaften können nach dem Erzeugen neue Werte bekommen. Der Zugriff von außen auf die Objekteigenschaften erfolgt über das bekannte Schema `instanzobjekt.member`. Wie gesagt sind Objekte in JavaScript (ECMAScript 3) jederzeit änderbar und erweiterbar.

```
var maunzi = new Katze('Maunzi', 'Perserkatze');
alert(maunzi.name + ' ist eine ' + maunzi.rasse);
maunzi.rasse = 'Siamkatze';
alert(maunzi.name + ' ist neuerdings eine ' + maunzi.rasse);
```

Da diese Eigenschaften von außen zugänglich und schreibbar sind, handelt es sich um **öffentliche Eigenschaften**.

Prototypische Objekte (Prototypen)

In den obigen Beispielen haben wir dem neuen Objekt direkt im Konstruktor Eigenschaften und Methoden hinzugefügt. Das bedeutet, dass diese Objekte mit jeder Instanz neu angelegt werden. Das ist *eine* Möglichkeit, wie dem Objekt Funktionalität hinzugefügt werden kann. Sie ist unter anderem dann notwendig, wenn Konstruktor-Parameter an das Instanzobjekt kopiert werden, wie es im Beispiel mit `this.name = name;` getan wird.

Jede Funktion hat eine `protot`
Eigenschaft, in der das prototy
Objekt steckt. Erweitern Sie di
vererben sich dessen Eigensch
alle Instanzen, die mit der Fun
erzeugt werden.

Mit einer Funktion ist immer ein **prototypisches Objekt (Prototyp)**verknüpft. Es handelt sich um ein gewöhnliches allgemeines JavaScript-Objekt, wie wir es auch mit `new Object()` oder dem `Object`-Literal `{}` anlegen können. Dieser Prototyp ist bei eigenen Funktionen anfangs leer.

Über die Eigenschaft `prototype` können wir ausgehend vom Funktionsobjekt auf den Prototypen zugreifen. Dieses Objekt können wir entweder erweitern oder mit einem eigenen Objekt ersetzen. Im folgenden Beispiel wird der Prototyp erweitert, indem eine Methode hinzugefügt wird:

```
function Katze () {}

Katze.prototype.miau = function () {
    alert("Miau!");
};

var maunzi = new Katze();
mauzi.miau();
```

Hier wird ein Funktionausdruck notiert und das Funktionsobjekt in `Katze.prototype.miau`gespeichert. Beim Prototypen wird also eine Eigenschaft namens `miau` angelegt. Darin steckt nun die neu angelegte Funktion. Wenn wir eine `Katze`-Instanz erzeugen, so besitzt sie eine `miau`-Methode.

Vererbung vom Prototypen zur Instanz

Durch den Aufruf von `new Katze` wird wie gesagt intern ein zunächst leeres `Object` erzeugt. Der Prototyp des Konstruktors, `Katze.prototype`, wird dabei in die sogenannte **Prototyp-Kette** (**Prototype Chain**) des Objektes eingehängt. Dies ist eine geordnete Liste mit Objekten, die abgearbeitet wird, wenn auf eine Eigenschaft des Objektes zugegriffen wird.

Der Prototyp steht »hinter« ein Objekt: Wird bei dem Objekt ei Eigenschaft nicht direkt gefun kann der Prototyp einspringen bereitstellen.

Ein konkretes Beispiel: Wenn wir den Ausdruck `maunzi.miau` schreiben, dann arbeitet der JavaScript-Interpreter die Prototyp-Kette ab, um die Eigenschaft namens `miau` zu finden und damit den Ausdruck aufzulösen. Die Prototyp-Kette von `maunzi` hat folgende Einträge:

1. `maunzi` – das Instanzobjekt selbst
2. `Katze.prototype` – der Prototyp für alle Objekte, die mit dem `Katze`-Konstruktor erzeugt wurden
3. `Object.prototype` – der Prototyp, der hinter allen Objekten steht

Das folgende Diagramm zeigt Objekte der Prototyp-Kette und listet deren Eigenschaften auf:

maunzi	
[[Prototype]]	Katze.prototype
constructor	Katze

Katze.prototype	
[[Prototype]]	Object.prototype
miau	function () { alert("Miau!"); }
constructor	Object

Object.prototype	
[[Prototype]]	null
constructor	Object
toString	[native Funktion]
toLocaleString	[native Funktion]
valueOf	[native Funktion]
hasOwnProperty	[native Funktion]
isPrototypeOf	[native Funktion]
propertyIsEnumerable	[native Funktion]

Wenn wir `maunzi.miau` notieren, dann wird nacheinander an diesen drei Objekten nach einer Eigenschaft mit diesem Namen gesucht.

Das Objekt `maunzi` besitzt selbst keine Eigenschaften bis auf `constructor`, welches auf die Funktion zeigt, mit dem es erzeugt wurde – das ist bekanntlich `Katze`. Intern besitzt `maunzi` einen Verweis auf seinen Prototyp, das ist `Katze.prototype`. Dieser Verweis wird in der unsichtbaren Eigenschaft `[[Prototype]]` gespeichert – lediglich in einigen JavaScript-Engines ist er über die Eigenschaft `__proto__` les- und schreibbar.

Über diesen Verweis schreitet der JavaScript-Interpreter zum Prototypen von `maunzi`,`Katze.prototype`. Dort wird er fündig, denn dort existiert eine Eigenschaft namens `miau`. Dies ist eine Funktion und sie kann mit dem Aufruf-Operator `(…)` ausgeführt werden.

Auf diese Weise stehen alle Eigenschaften eines Prototypen, im Beispiel`Katze.prototype`, auch beim Instanzobjekt zur Verfügung, im Beispiel`maunzi`. Dies ist das ganze Geheimnis hinter der **prototypischen Vererbung**. Wenn bei einem Objekt selbst die angeforderten Eigenschaften nicht gefunden wurde, so dienen die Objekte in der Prototyp-Kette als Fallback. Man spricht von einer **Delegation** (Übertragung, Weitergabe). Das Objekt gibt die Anfrage an seine Prototypen weiter.

Prototypische Vererbung ist da Delegieren von Funktionalität einem Objekt zum anderen. Da prinzipiell nicht an Konstrukto Instanzen gebunden, über dies bloß der Prototyp-Verweis ang

Prototypische Vererbung funktioniert grundlegend anders als klassenbasierte Vererbung, denn JavaScript ist eine äußerst dynamische Sprache. Es gibt keine Klassen, die einmal deklariert werden und nach der Kompilierung unveränderlich sind. Über die Prototyp-Kette erben gewöhnliche Objekte von gewöhnlichen Objekten. Jedes Objekt kann der Prototyp eines anderen Objektes werden und »einspringen«, wenn es die angeforderte Eigenschaft nicht selbst bereitstellen kann.

Alle beteiligten Objekte einschließlich der Prototypen können zur Laufzeit beliebig verändert werden. Ein Prototyp ist also im Gegensatz zu einer Klasse kein fester Bauplan für immer gleiche Instanzen, sondern selbst ein beliebiges Objekt, an das Eigenschaftsanfragen delegiert werden. Deshalb ist der Begriff »Instanz« für das Objekt, welches `new Katze` erzeugt, letztlich irreführend. Wo es keine Klassen als rein abstrakten Bauplan gibt, sondern bloß flexible Objekte mittels Konstruktoren erzeugt werden und von Prototypen erben, so trifft dieser Begriff die Sache nicht. Er wird hier trotzdem der Einfachheit halber verwendet.

Prototypen verstehen: Die Glasplatten-Metapher

Eine hervorragende Veranschaulichung von Prototypen hat Robin Debreuil ausgearbeitet. Er schrieb 2001 ein Tutorial über objektorientierte Programmierung mit ActionScript 1.0 in Flash 5. ActionScript war damals eine Sprache, die auf ECMAScript 3 aufbaute und damit in den Grundzügen mit JavaScript identisch war. Die Sprache ActionScript ist mittlerweile von prototypenbasierter auf klassenbasierte Objektorientierung umgestiegen – für JavaScript sind diese Erklärungen aber immer noch gültig und aufschlussreich.

Die Metapher beschreibt Objekte in der Prototyp-Kette als beklebte Glasplatten. Auf jeder Glasplatte sind farbige Zettel an bestimmten Positionen aufgeklebt. Diese Zettel entsprechen den Objekteigenschaften, die Positionen entsprechen Eigenschaftsnamen. Die Instanz selbst verfügt über ein paar Zettel, sein Prototyp und dessen Prototyp über weitere. Diese können auch an denselben Stellen kleben.

Die Funktionalität, über die die Instanz verfügt, ist nun eine Summe der Zettel der drei Glasplatten: In der Metapher werden die Glasplatten übereinandergelegt. Da sie durchsichtig sind, schimmern durch die Lücken die Zettel auf den darunterliegenden Platten durch. Das Gesamtbild, das sich so ergibt, setzt sich aus den Zetteln der drei Platten zusammen.

Das Objekt ist demnach ein Mosaik, das sich aus den eigenen sowie fremden Eigenschaften zusammensetzt. Welches Objekt in der Prototyp-Kette nun eine gesuchte Eigenschaft bietet, ist unwichtig. An der gesuchten Stelle auf dem Glas ist ein Zettel sichtbar – in der Grafik z.B. oben links, des entspricht einem Eigenschaftsnamen, z.B. `eins`.

Die Metapher zeigt auch, dass Objekte in der Prototyp-Kette gleiche Eigenschaften bieten können. An der Stelle oben in der Mitte klebt ein gelber Zettel auf der Instanz-Glasplatte, aber auch ein orangener auf der, die `Konstruktor.prototype` darstellt. Beim Übereinanderlegen ist nur der gelbe Zettel der oben liegenden Instanz-Glasplatte sichtbar, der orangene wird überdeckt. Übertragen heißt das: Eine Eigenschaft kann am Instanzobjekt definiert werden, und schon überschreibt sie eine gleichnamige Eigenschaft am `prototype`-Objekt des Konstruktors.

Diese Art der Vererbung nennt man **Differential Inheritance**. Im Gegensatz zur klassenbasierten Vererbung werden beim abgeleiteten Objekt (der Instanz) keine Eigenschaften erzeugt. Die Instanz ist keine Kopie des Prototypen, die Instanz kann sogar leer sein, wie es Katzen-Beispiel der Fall ist. Erst wenn sich die Instanz vom Prototypen unterscheidet, wird bei der Instanz eine Eigenschaft angelegt, die einen anderen Wert als die des Prototypen besitzt. In der Glasplatten-Metapher bedeutet dies: An den Stellen, in denen die Instanz dem Prototyp gleicht, ist die Platte durchsichtig – sie delegiert . Wo sie sich unterscheidet, besitzt sie einen eigenen, andersfarbigen Zettel.

> Bei prototypischer Vererbung
> nur sich unterscheidende
> Eigenschaften beim abgeleitet
> Objekt gesetzt.

Nehmen wir an, dass hiesige Katzen meistens von der Rasse »Europäisch Kurzhaar« sind. Anstatt dies jedes Mal beim Erzeugen anzugeben, legen wir die Eigenschaft beim Prototypen an:

```
function Katze () {}
Katze.prototype.rasse = "Europäisch Kurzhaar";

var maunzi = new Katze();
alert(maunzi.rasse);
```

Greifen wir auf `maunzi.rasse` zu, so wird die `rasse`-Eigenschaft beim Prototypen gefunden. Denn die relevanten Objekte sehen so aus (`Object.prototype` wurde ausgeblendet):

maunzi	
[[Prototype]]	Katze.prototype
constructor	Katze

Katze.prototype	
[[Prototype]]	Object.prototype
rasse	"Europäisch Kurzhaar"
constructor	Object

Wenn wir nun eine Katze mit abweichender Rasse erzeugen wollen, so legen wir eine gleichnamige Eigenschaft bei der Instanz an, die die Eigenschaft des Prototypen verdeckt:

```
var maunzi = new Katze();
```

```
alert(maunzi.rasse); // Derzeit noch »Europäisch Kurzhaar« - vererbt vom Prototypen

maunzi.rasse = "Perser";
alert(maunzi.rasse); // Jetzt »Perser« - eigene Eigenschaft
```

Daraufhin besitzt die Instanz eine eigene, abweichende Eigenschaft:

maunzi	
[[Prototype]]	Katze.prototype
rasse	"Perser"
constructor	Katze

Katze.prototype	
[[Prototype]]	Object.prototype
rasse	"Europäisch Kurzhaar"
constructor	Object

Private Objekte anstatt private Eigenschaften

Viele klassenbasierte Sprachen erlauben es, die Sichtbarkeit von Instanzeigenschaften festzulegen und unterscheiden beispielsweise zwischen öffentlichen und privaten Eigenschaften.

In JavaScript (ECMAScript 3) gibt es keine Möglichkeit, gewisse Eigenschaften eines Objektes als privat zu deklarieren, sodass sie ausschließlich in Methoden des Objekt zur Verfügung stehen. Sobald wir einem Objekt eine Eigenschaft hinzufügen, ist diese auch überall dort verfügbar, wo das Objekt verfügbar ist. Ebenso ist es nicht möglich, eine Eigenschaft als nicht überschreibbar und nicht löschbar zu deklarieren. Dies ist erst in ECMAScript 5 möglich, welches derzeit noch nicht breit unterstützt wird.

```
function Katze () {}

Katze.prototype.pfoten = 4;
Katze.prototype.miau = function () {
   alert("Miau!");
};

var maunzi = new Katze();
var schnucki = new Katze();

// Überschreibe vom Prototyp vererbte Eigenschaft,
// indem eine gleichnamige Eigenschaft bei der Instanz erzeugt wird:
maunzi.pfoten = 5;
alert('Maunzi hat nun ' + maunzi.pfoten + ' Pfoten.');

// Überschreibe Methode des Prototyps:
Katze.prototype.miau = function () {
   alert("Wau, wau!");
};
schnucki.miau();
```

Das obige Beispiel zeigt, wie ein Instanzobjekt und auch der Prototyp nachträglich verändert werden können: Plötzlich hat Maunzi fünf Pfoten und alle Katzen sagen »wau« anstatt »miau«. Dies ist sowohl ein Defizit von ECMAScript 3 als auch eine Stärke: Prototypen sind nicht abgeschlossen und Objekte immer erweiterbar.

Wie gesagt gibt es keine privaten Eigenschaften im Wortsinn, auch wenn manche diesen Begriff auch auf JavaScript anwenden. Denn wenn ein Objekt an der Instanz hängt, ist es in ECMAScript 3 auch notwendig nach außen sichtbar und unkontrolliert überschreibbar. Wir können jedoch einen Trick anwenden, den wir bereits vom Revealing Module Pattern kennen: In einem Funktions-Scope notieren wir lokale Variablen und zudem die öffentlichen Methoden des Objektes. Die öffentlichen Methoden haben auf erstere Zugriff, weil sie im selben Scope erzeugt wurden und damit Closures sind. Daher handelt es sich um sogenannte **priviligierte Methoden**.

Private Eigenschaften gibt es n
wir können stattdessen einen
Funktions-Scope definieren, de
private Variablen die Instanzm
einschließen.

Da der Konstruktor bereits einen Funktions-Scope bereitstellt, nutzen wir diesen für private Objekte. Damit die Methoden auf die privaten Objekte Zugriff haben, müssen sie im Konstruktor erzeugt und dürfen nicht über den Prototyp definiert werden.
Über `this` werden sie ans Instanzobjekt gehängt.

```
function Katze (name) {
    // --- Private Objekte
    // Private Variablen
    var pfoten = 4;
    var gestreichelt = 0;
```

```
    // name ist ebenfalls eine private Variable

    // Private Funktionen
    function miau () {
        alert(name + ' macht miau!');
    }

    // --- Öffentliche (priviligierte) Eigenschaften
    this.name = name;
    // Öffentliche Methoden
    this.kitzeln = function () {
        alert(name + ' hat ' + pfoten + ' kitzlige Pfoten.');
        miau();
    };
    this.streicheln = function () {
        gestreichelt++;
        miau();
    };
}

var maunzi = new Katze('Maunzi');
maunzi.kitzeln();
maunzi.streicheln();

alert('maunzi.name: ' + maunzi.name);
// pfoten ist keine Objekt-Eigenschaft, also von außen unzugänglich:
alert('maunzi.pfoten: ' + maunzi.pfoten);
```

Der Konstruktor nimmt hier den Parameter `name` entgegen. Dieser ist automatisch eine lokale Variable. Zusätzlich werden zwei lokalen Variablen (`pfoten`, `gestreichelt`) sowie eine lokale Funktion (`miau`) angelegt. Der leeren Instanz werden eine Eigenschaft (`name`) und zwei öffentliche Methoden (`kitzeln`, `streicheln`) angehängt, die als verschachtelte Funktionsausdrücke notiert werden.

Nach dem Anlegen einer `Katze` mit dem Namen *Maunzi* werden die beiden Methoden aufgerufen. Sie haben Lese- und Schreibzugriff auf die privaten Objekte und können auch die private Funktion ausführen, welche von außen nicht zugänglich sind.

Wie das Beispiel zeigt, können auch private Funktionen angelegt werden. Private Funktionen können zum Beispiel interne, in verschiedenen Methoden verwendete Helfer sein. Sie entsprechen *privaten Methoden* in klassenbasierten Sprachen. Dieser Begriff ist auf JavaScript nicht anwendbar, da es sich eben nicht um Methoden des Instanzobjekt handelt. Sie sind lediglich in den tatsächlichen Instanzmethoden verfügbar, da diese Zugriff auf die Variablen des Konstruktor-Scopes haben.

> Funktions-Scopes und Closures
> der Schlüssel zu privaten Objek
> sowohl bei Konstruktoren/Inst
> und dem Revealing Module Pat

Nachteile von privaten Objekten

Der gravierende Unterschied zu den vorigen Beispielen ist, dass die Nutzung des Prototyps und damit die Vererbung wegfällt. Anstatt die Methoden einmal am Prototyp zu erzeugen, werden sie bei jeder Instanz im Konstruktor von neuem angelegt. Heraus kommt folgende Prototyp-Kette:

maunzi	
[[Prototype]]	Katze.prototype
name	'Maunzi'
kitzeln	function () { … }
streicheln	function () { … }

Katze.prototype	
[[Prototype]]	Object.prototype
constructor	Object

Object.prototype	
[[Prototype]]	null
constructor	Object
toString	[native Funktion]
toLocaleString	[native Funktion]
valueOf	[native Funktion]
hasOwnProperty	[native Funktion]
isPrototypeOf	[native Funktion]
propertyIsEnumerable	[native Funktion]

Der Prototyp ist demnach leer, die Eigenschaften hängen direkt an der Instanz und werden nicht vererbt. Dies hat verschiedene Konsequenzen:

Ein Überschreiben oder Löschen dieser Eigenschaften über den Prototyp ist nicht möglich. Ein nachträgliches Erweitern über Prototyp möglich, diese Methoden sind allerdings nicht priviligiert, haben also keinen Zugriff auf private Objekte. Denn sie wurden nicht im Konstruktor definiert und schließen die privaten Objekte nicht ein. Aus demselben Grund ist kein

> Wenn Methoden über den Prot
> anstatt im Konstruktor erzeugt
> werden, wird Speicher eingesp
> Dafür ist das Auflösen von
> Eigenschaften über die Prototy
> etwas langsamer, da an mehre

nachträgliches Hinzufügen von privaten Objekten möglich.

Objekten gesucht wird.

Mit jedem Erzeugen einer `Katze`-Instanz werden alle Eigenschaften von neuem erzeugt. Die Instanzen teilen sich ihre Eigenschaften nicht mit anderen`Katze`-Instanzen. Dies wirkt sich negativ auf die Performance aus: Das Anlegen der Eigenschaften kostet Zeit und Arbeitsspeicher. Werden z.B. zehn Katzen instantiiert, so werden 20 Funktionsobjekte erzeugt (zehn mal `kitzeln` und `streicheln`). Würden die Instanzen diese Methoden vom Prototyp erben, so müssten sie nur einmal am Prototyp erzeugt werden.

Wozu Kapselung gut ist und wann sie nötig ist

Kapselung in objektorientiertem JavaScript hat zwei Bedeutungen: Zum einen wird das Programm so strukturiert, dass es möglichst nicht mit anderen Scripten in Konflikt kommt. Dafür muss man in JavaScript selbst Sorge tragen, denn alle Scripte teilen sich das globalen `window`-Objekt. Zum anderen bedeutet Kapselung die Trennung zwischen öffentlichen und privaten Objekten. Auf die öffentlichen können andere Scripte von außen zugreifen, die privaten sind diesen verborgen.

Die erste Art der Kapselung ist in jedem Fall sinnvoll, eine effektive Kapselung der zweiten Art bringt jedoch Nachteile mit sich. Wann also brauchen wir sie und in welchem Umfang?

Das Konzept der Kapselung in der objektorientierten Programmierung hat den Zweck, zwischen einer öffentliche API und der internen Implementierung zu unterscheiden. Sie hilft in aller erster Linie dem Programmierer, gut strukturierte, wiederverwendbare Software zu schreiben.

Die API, also das Set an öffentlichen Methoden, wird dokumentiert und von anderen Programmierern angesteuert. Sie bleibt idealerweise über mehrere Programmversionen gleich. Sie kann natürlich erweitert und beizeiten durch eine leistungsfähigere und flexiblere Schnittstelle ergänzt werden.

Die privaten Objekte umfassen alle Variablen und Methoden, die nur intern Gebrauch finden und die tatsächliche Implementierung strukturieren. Ein bloßer Anwender des Scriptes muss diese nicht kennen. Von Version zu Version kann sich die interne Umsetzung bei gleichbleibender Funktionalität ändern, damit auch die privaten Objekte.

Ein zweiter Grund für Kapselung ist die effektive Sicherheit: Ein fremdes Script soll die Interna nicht lesen und manipulieren können. Dies ist in anderen Sprachen sehr wichtig. Manche JavaScript-Kenner argumentieren, dass der Aspekt der tatsächlichen Unsichtbarkeit für JavaScript nachrangig ist:

- Wenn ein Script ein anderes manipulieren will, dann ist das auf die eine oder andere Art möglich, weil ECMAScript 3 so dynamisch ist. Erst ECMAScript 5 wird es ermöglichen, einzelne Objekteigenschaften oder ganze Objekte »einzufrieren« und vor Manipulation zu schützen.
- Mit Tricks ist es bei einigen JavaScript-Interpretern möglich, auf einen privaten Funktions-Scope zuzugreifen.
- Es kann durchaus von Vorteil sein, die Interna eines Scriptes lesen, abändern und erweitern zu können. Das gehört zu den Features von JavaScript.

Pseudo-private Objekte

Aus diesen Gründen kann es ausreichen, Kapselung lediglich als Strukturierungskonzept zu verstehen. Es reicht dann, mit **pseudo-privaten Objekten** zu arbeiten. Diese sind nicht effektiv vor dem Zugriff von außen geschützt, finden jedoch in der Regel nur intern Verwendung.

Ein einfacher `Object`-Literal bietet keine effektive Kapselung, d.h. alle Eigenschaften sind von außen sichtbar und änderbar. Kapselung als Konzept lässt sich damit trotzdem umsetzen, indem klar zwischen öffentlichen und (pseudo-)privaten Eigenschaften unterschieden wird. Eine Konvention ist etwa, die privaten Eigenschaften mit einem `_` (Unterstrich) beginnen zu lassen. Ein entsprechendes Modul könnte folgendermaßen aussehen:

```
var Modul = {
    // Öffentliche Eigenschaften
    öffentlicheMethode : function () {
        alert(this._privateMethode());
    },
    // Pseudo-private Eigenschaften
```

```
    _privateEigenschaft : 1;
    _privateMethode : function () {
        this._privateEigenschaft++;
        return this._privateEigenschaft;
    }
};
Module.öffentlicheMethode();
```

Dasselbe bei einem Prototypen:

```
function Konstruktor () {}

// Öffentliche Eigenschaften
Konstruktor.prototype.öffentlicheMethode = function () {
   alert(this._privateMethode());
};

// Pseudo-private Eigenschaften
Konstruktor.prototype._privateEigenschaft = 1;
Konstruktor.prototype._privateMethode = function () {
   this._privateEigenschaft++;
   return this._privateEigenschaft;
};

var instanz = new Konstruktor();
instanz.öffentlicheMethode();
```

Wie gesagt, technisch gesehen handelt es sich bei diesen pseudo-privaten Eigenschaften um ganz normale Eigenschaften, die sich in puncto Sichtbarkeit und Schreibbarkeit nicht von den sogenannten öffentlichen unterscheiden. Der Unterstrich im Namen hat keine Bedeutung für den JavaScript-Interpreter, er ist lediglich eine Namenskonvention.

Organisation von JavaScripten: Objektverfügbarkeit und `this`-Kontext

Bedeutung von this

Bei allen drei vorgestellten Techniken – `Object`-Literalen, Revealing Module Pattern und Konstruktoren/Prototypen – haben wir mit `this` gearbeitet, um auf das Modul bzw. die Instanz zuzugreifen. Nun wollen wir uns näher der Funktionsweise von `this` zuwenden.

`this` ist ein Schlüsselwort, das zu einem Wert aufgelöst wird und in der Regel auf ein Objekt zeigt. Worauf es zeigt, hängt ganz vom Kontext ab.

Notieren wir `this` im globalen Scope, so zeigt es bloß auf `window`, das globale Objekt:

```
alert(this); // Ergibt [object Window] oder ähnliches
```

Dasselbe gilt, wenn `this` in einer Funktion verwendet wird, die ganz normal über `funktion()` aufgerufen wird:

```
function zeigeThis () {
    alert(this); // ergibt ebenfalls [object Window]
}
zeigeThis();
```

In diesen beiden Fällen bietet `this` wenig Nutzen, denn wir könnten genauso `window` schreiben.

`this` wird in folgendem Fall interessant: Eine Funktion hängt an einem Objekt, ist also eine Methode dessen. Wenn wir die Funktion nun über das Schema `objekt.funktion()` aufrufen, dann zeigt `this` innerhalb der Funktion auf `objekt`.

`this` zeigt standardmäßig auf
globale Objekt `window`. Innerh
Methode, die
über`objekt.funktion()` au
wird, zeigt es jedoch auf `objek`

Ein einfaches Modul mit einem `Object`-Literal:

```
var Modul = {
    eigenschaft : "wert",
    methode : function () {
        alert("methode wurde aufgerufen\n" +
              "this.eigenschaft: " + this.eigenschaft);
    }
};
Modul.methode();
```

`this` zeigt innerhalb der Methode auf das Objekt `Modul`. Wir könnten alternativ`Modul.eigenschaft` schreiben, was auf dasselbe herauskäme. `this` hat jedoch den Vorteil, dass es unabhängig vom aktuellen Modulnamen ist. Wenn dieser später geändert wird und die Funktion verschoben wir, so müssen nicht alle Verweise angepasst werden.

Bei Konstruktoren und Prototypen ist `this` unersetzlich:

```
function Katze (name) {
    this.name = name;
}
Katze.prototype = {
    pfoten : 4,
    zeigePfoten : function () {
        alert("Die Katze zeigt ihre " + this.pfoten + " Pfoten.");
    }
};
var maunzi = new Katze('Maunzi');
maunzi.zeigePfoten();
var schnucki = new Katze('Schnucki');
schnucki.zeigePfoten();
```

`this` zeigt innerhalb des Konstruktors und der `zeigePfoten`-Methode auf die Instanz. Dies ist einmal `maunzi` und einmal `schnucki`, deshalb müssen wir hier `this` verwenden.

Methoden in anderen Kontexten ausführen

Der Zugriff auf das Modul bzw. auf die Instanz über `this` ist zum Teil unerlässlich. Damit `this` auf das gewünschte Objekt zeigt, müssen beide Kriterien erfüllt sein: Die Funktion hängt an dem Objekt als Unterobjekt **und** sie wird über das Schema `objekt.funktion()` aufgerufen. Alleine durch diese Aufrufweise wird die `this`-Verbindung hergestellt.

`this` ist in der OOP äußerst pr
Allerdings verweist `this` nur
Aufrufweise `objekt.funktio`
das gewünschte Objekt. Der Ve
geht bei anderen Aufrufweisen
der funktionalen Natur von Jav
liegen, verloren.

Dieser Bezug kann jedoch verloren gehen, wenn die Funktion außerhalb dieses Objektkontextes ausgeführt wird. Dies passiert vor allem in folgenden Fällen:

1. Beim Event-Handling, wenn die Funktion als Event-Handler registriert wird. Beim Unobtrusive JavaScript ist es üblich, dass Methoden eines Moduls oder einer Instanz als Event-Handler dienen (siehe Grundlagen zur Ereignisverarbeitung). `this` zeigt in Handler-Funktionen auf das Elementobjekt, bei dem das Ereignis verarbeitet wird – siehe this beim Event-Handling. Dadurch werden die Methoden außerhalb des Modul- bzw. Instanzkontextes ausgeführt.
2. Beim Aufrufen der Funktion mit `setTimeout` oder `setInterval`. Die verzögert bzw. wiederholt ausgeführte Funktion verliert den Bezug zum Ursprungsobjekt, denn `this` verweist darin auf das globale Objekt `window`. In vielen Fällen ist der Zugriff auf das Modul bzw. die Instanz notwendig.
3. Bei der Übergabe einer Funktion als Parameter (z.B. als Callback-Funktion), beim Speichern in einer Variablen und dergleichen. In diesen Fällen zeigt gibt es oftmals keinen spezifischen Kontext, sodass `this` als Fallback auf `window` zeigt.

`this`-Problem bei einfachen Modulen

Das folgende Beispiel demonstriert das Problem im Falle eines einfachen Moduls mit dem`Object`-Literal:

```
var Modul = {
    eigenschaft : "Eigenschaftswert",
    start : function () {
        // Funktioniert:
        alert("start wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
        setTimeout(this.verzögert, 100);
        document.getElementById("button").onclick = this.handler;
    },
    verzögert : function () {
        // Fehler: this verweist auf window
        alert("verzögert wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
    },
    handler : function (e) {
        // Fehler: this verweist auf das Element, dem der Event-Handler anhängt
        alert("handler wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
    }
};
Modul.start();
```

Das zugehörige HTML:

```
<button id="button">Button, der auf Klick reagiert</button>
```

`this`-Problem bei Prototypen und Instanzmethoden

Dasselbe mit einem Konstruktor, einem Prototyp und einer Instanz:

```
function Konstruktor () {}
Konstruktor.prototype = {
    eigenschaft : "Eigenschaftswert",
    start : function () {
        // Funktioniert:
        alert("start wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
        setTimeout(this.verzögert, 100);
        document.getElementById("button").onclick = this.handler;
    },
    verzögert : function () {
        // Fehler: this verweist auf window
        alert("verzögert wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
    },
    handler : function (e) {
        // Fehler: this verweist auf das Element, dem der Event-Handler anhängt
        alert("handler wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
    }
};
var instanz = new Konstruktor();
instanz.start();
```

In beiden Fällen werden Objektmethoden als Event-Handler verwendet (`handler`) sowie mit`setTimeout` aufgerufen (`verzögert`). In den `start`-Methoden gelingt der Zugriff über `this` noch. In der `verzögert`-Methode zeigt `this` jedoch nicht mehr auf das richtige Objekt, sondern auf `window`. In der `handler`-Methode, welche beim Klicken auf den Button ausgeführt wird, enthält `this` zwar eine wertvolle Information, aber auch hier geht der Bezug zum Modul bzw. zur Instanz verloren.

Die Lösung dieses Problems ist kompliziert und führt uns auf eine zentral wichtige, aber auch schwer zu meisternde Eigenheit der JavaScript-Programmierung, die im Folgenden vorgestellt werden soll.

Einführung in Closures

Eine **Closure** ist allgemein gesagt eine Funktion, die in einer anderen Funktion notiert wird. Diese verschachtelte, innere Funktion hat Zugriff auf die Variablen des Geltungsbereiches (Scopes) der äußeren Funktion – und zwar über die Ausführung der äußeren Funktion hinaus.

Verschachtelte Funktionen hab
Zugriff auf die lokalen Variable
äußere Funktion – und zwar au
der Abarbeitung der äußeren F
Sie konservieren die Variablen

Durch dieses **Einschließen** der Variablen kann man bestimmte Objekte in Funktionen verfügbar machen, die darin sonst nicht oder nur über Umwege verfügbar wären. Closures werden damit zu einem Allround-Werkzeug in der fortgeschrittenen JavaScript-Programmierung. Wir haben Closures bereits verwendet, um private Objekte zu erreichen.

Dieses Beispiel demonstriert die Variablen-Verfügbarkeit bei verschachtelten Funktionen:

```
function äußereFunktion  () {
    // Definiere eine lokale Variable
    var variable = "wert";
    // Lege eine verschachtelte Funktion an
    function innereFunktion () {
        // Obwohl diese Funktion einen eigenen Scope mit sich bringt,
        // ist die Variable aus dem umgebenden Scope hier verfügbar:
        alert("Wert der Variablen aus der äußeren Funktion: " + variable);
    }
    // Führe die eben definierte Funktion aus
    innerfunktion();
}
äußereFunktion();
```

Das Beispiel zeigt, dass die innere Funktion Zugriff auf die Variablen der äußeren Funktion hat. Der entscheidende Punkt bei einer Closure ist jedoch ein anderer:

Normalerweise werden alle lokalen Variablen einer Funktion aus dem Speicher gelöscht, nachdem die Funktion beendet wurde. Eine Closure aber führt dazu, dass die Variablen der äußeren Funktion nach deren Ausführung nicht gelöscht werden, sondern im Speicher erhalten bleiben. Die Variablen stehen der inneren Funktion weiterhin über deren ursprüngliche Namen zur Verfügung. Die Variablen werden also*eingeschlossen* und konserviert – daher der Name »Closure«.

Auch lange nach dem Ablauf der äußeren Funktion hat die Closure immer noch Zugriff auf deren Variablen. Vorausgesetzt ist, dass die Closure woanders gespeichert wird und dadurch zu einem späteren Zeitpunkt ausgeführt werden kann. Im obigen Beispiel ist die innere Funktion nur eine lokale Variable, die zwar Zugriff auf die Variablen der äußeren Funktion hat, aber bei deren Beendigung selbst verfällt.

Eine Möglichkeit, die innere Funktion zu speichern, ist das Registrieren als Event-Handler. Dabei wird das Funktionsobjekt in einer Eigenschaft (hier `onclick`) eines Elementobjektes gespeichert und bleibt damit über die Ausführung der äußeren Funktion hinweg erhalten:

```
function äußereFunktion  () {
    var variable = "wert";
    // Lege eine verschachtelte Funktion an
    function closure {
        alert("Wert der Variablen aus der äußeren Funktion: " + variable);
    };
    // Speichere die Closure-Funktion als Event-Handler
    document.getElementById("button").onclick = closure;
}
äußereFunktion();
```

Der zugehörige Button im HTML:

```
<button id="button">Button, der auf Klick reagiert</button>
```

Bei einem Klick auf den Button wird die Closure als Event-Handler ausgeführt. `äußereFunktion`wird schon längst nicht mehr ausgeführt, aber `variable` wurde in die Closure eingeschlossen.

Zusammengefasst haben wir folgendes Schema zur Erzeugung einer Closure:

1. Beginn der Ausführung der äußeren Funktion
2. Lokale Variablen werden definiert
3. Innere Funktion wird definiert
4. Innere Funktion wird außerhalb gespeichert, sodass sie erhalten bleibt
5. Ende der Ausführung der äußeren Funktion
6. Unbestimmte Zeit später: Innere Funktion (Closure-Funktion) wird ausgeführt

Anwendung von Closures: Zugriff auf das Modul bzw. die Instanz ermöglichen

Wie helfen uns Closures nun beim `this`-Problem weiter?

Module: Verzicht auf `this` zugunsten des Revealing Module Patterns

Gegenüber dem einfachen `Object`-Literal bietet das Revealing Module Pattern bereits die nötige Infrastruktur, um das Problem zu lösen. Um private Objekte zu erreichen, benutzt das Revealing Module Pattern eine Kapselfunktion, in der weitere Funktionen notiert sind. Die inneren Funktionen sind bereits Closures. Daher liegt eine mögliche Lösung darin, vom einfachen `Object`-Literal auf das Revealing Module Pattern umzusteigen.

Beim Revealing Module Patter
ist`this` unnötig, denn alle Fun
sind Closures und schließen di
benötigten Variablen ein.

Das Revealing Module Pattern trennt zwischen privaten Objekten und der öffentliche Schnittstelle (API). Letztere ist ein `Object`-Literal, der aus der Kapselfunktion zurückgegeben wird. Dieses `Object`enthält verschachtelte Methoden, die als Closures die privaten Objekte einschließen. Zur Wiederholung:

```
var Modul = (function () {
    // Private Objekte
    var privateVariable = "privat";
    // Öffentliche API
    return {
        öffentlicheMethode : function () {
            alert(privateVariable);
        }
    };
})();
Modul.öffentlicheMethode();
```

Die Funktionen haben in jedem Fall Zugriff auf die privaten Objekte, auch wenn sie z.B. durch Event-Handling oder `setTimeout` aus dem Kontext gerissen werden. Eine kleine Änderung ist jedoch nötig, damit öffentliche Methoden sich gegenseitig sowie private Funktionen öffentliche Methoden aufrufen können. Anstatt die öffentliche API-Objekt direkt hinter `return` zu notieren, speichern wir es zuvor in einer Variable. Diese wird von allen verschachtelten Funktionen eingeschlossen.

```
var Modul = (function () {
    // Private Objekte
    var privateVariable = "privat";
    function privateFunktion () {
        alert("privateFunktion wurde verzögert aufgerufen\n"+
            "privateVariable: " + privateVariable);
        // Rufe öffentliche Methode auf:
        api.end();
    }
    // Öffentliche API, gespeichert in einer Variable
    var api = {
        start : function () {
            alert("Test startet");
            // Hier würde this noch funktionieren, wir nutzen trotzdem api
            setTimeout(api.öffentlicheMethode, 100);
        }
        öffentlicheMethode : function () {
            alert("öffentlicheMethode wurde verzögert aufgerufen");
            setTimeout(privateFunktion, 100);
        }
        ende : function () {
            alert("Öffentliche ende-Methode wurde aufgerufen. Test beendet");
        }
    };
    return api;
})();
Modul.start();
```

Der Code gibt folgende Meldungen aus:

```
Test startet
öffentlicheMethode wurde verzögert aufgerufen
privateFunktion wurde verzögert aufgerufen
privateVariable: privat
Öffentliche ende-Methode wurde aufgerufen. Test beendet
```

Dieses Beispiel zeigt, wie öffentliche und private Methoden einander aufrufen können. Auf `this`kann verzichtet werden, denn alle benötigten Variablen werden durch Closures eingeschlossen. Infolgedessen stellt die Nutzung von `setTimeout` kein Problem dar.

Die privaten Objekte sind direkt über ihre Variablennamen verfügbar, die Eigenschaften der öffentlichen API indirekt über die Variable `api`.

Konstruktoren/Instanzen: Methoden im Konstruktor verschachteln

Wir haben zwei Möglichkeit kennengelernt, dem Instanzobjekt Eigenschaften zuzuweisen. Zum einen, indem wir sie im Konstruktor über `this`anlegen. Zum anderen, indem wir einen Prototypen definieren und die Instanz davon erbt. Der Weg über den Prototyp hat Performance-Vorteile, der Weg über den Konstruktor erlaubt private Objekte.

Die Methoden werden im Kons
verschachtelt. Sie schließen ei
Variable ein, die auf die Instan
verweist und die wir als Ersatz
zu `this`verwenden..

Private Objekte funktionieren letztlich über Closures und bieten die Möglichkeit, das `this`-Problem zu umgehen: Im Konstruktor wird eine lokale Variable als Referenz auf das Instanzobjekt `this` angelegt. Diese heißt üblicherweise `thisObject`, `that` oder `instance`. Alle Methoden, die der Instanz im Konstruktor hinzugefügt werden, schließen diese Variable ein. Sie ist darin auch dann verfügbar, wenn sie als Event-Handler oder mit Verzögerung in einem anderen Kontext ausgeführt werden. Folgendes Beispiel demonstriert beide Fälle:

```
function Konstruktor () {
    // Referenz auf das Instanzobjekt anlegen
    var thisObject = this;
    // Weitere private Objekte
    var privateVariable = "privat";

    // Öffentliche Eigenschaften
    this.eigenschaft = "wert";

    this.start = function () {
        alert("start() wurde aufgerufen\n" +
            "Instanz-Eigenschaft: " + thisObject.eigenschaft);
        setTimeout(thisObject.verzögert, 500);
    };

    this.verzögert = function () {
        alert("verzögert() wurde aufgerufen\n" +
            "Instanz-Eigenschaft: " + thisObject.eigenschaft);
    };

    this.handler = function () {
        alert("handler wurde aufgerufen\n" +
            "Element, das den Event behandelt: " + this + "\n" +
            "Instanz-Eigenschaft: " + thisObject.eigenschaft);
    };

    // Hier im Konstruktor kann this noch verwendet werden
    document.getElementById("button").onclick = this.handler;
}

var instanz = new Konstruktor();
instanz.start();
```

Der zugehörige Button-Code lautet wieder:

```
<button id="button">Button, der auf Klick reagiert</button>
```

Wichtig ist hier die Unterscheidung zwischen `this` und `thisObject`. `this` zeigt in den drei Methoden `start`, `verzögert` und `handler` auf drei unterschiedliche Objekte. In `start` zeigt es auf das Instanzobjekt `instanz`,

in `verzögert` auf `window` und in `handler` auf das Button-Element.`thisObject` hingegen ist die eingeschlossene Variable, die auf das Instanzobjekt zeigt – und zwar in allen drei Methoden.

Dank Closures können wir zum Zugriff auf die Instanz auf das uneindeutige `this` verzichten. Stattdessen nutzen wir die eigene Variable `thisObject`.

Function Binding: Closures automatisiert erzeugen

Die gezeigte Verschachtelung ist eine effektiver, aber folgenschwerer Trick, um die Verfügbarkeit von Objekten zu gewährleisten. Sie liegt nahe, wenn sowieso mit privaten Objekten gearbeitet wird und deshalb alle Modul- bzw. Instanzmethoden verschachtelt werden. Sie funktioniert nicht bei einfachen`Object`-Literalen und bei der Nutzung von Prototypen. Glücklicherweise erlaubt die funktionale Natur von JavaScript, Funktionen und damit Closures zur Laufzeit anzulegen und den `this`-Kontext einer Funktion bei ihrem Aufruf festzulegen.

Mit Function Binding lassen sic
Bedarf Closures erzeugen, die
den`this`-Kontext einer Funkti
festlegen. Heraus kommt eine
Wrapper-Funktion, die die
ursprüngliche Funktion im
gewünschten Kontext aufruft.

`this`-Kontext erzwingen mit `call` und `apply`

Da Funktionen in JavaScript Objekte erster Klasse sind, können sie selbst Methoden besitzen. Zwei der vordefinierten Methoden von Funktionsobjekten sindcall und apply. Diese rufen die zugehörige Funktion auf und erlauben es zusätzlich, den Kontext beim Aufruf einer Funktion explizit anzugeben. Das Objekt, auf das `this` innerhalb der Funktion zeigt, wird nicht mehr nach den üblichen Regeln bestimmt. Stattdessen zeigt `this` auf das Objekt, das `call` bzw. `apply`übergeben wird.

Die Methoden `call` und `apply` von Funktionsobjekten erlauben das Erzwingen des Kontextes beim Funktionsaufruf.

Auf diese Weise können wir jede beliebige Funktion im Kontext eines beliebigen Objektes ausführen, auch ohne dass die Funktion am angegebenen Objekt hängt:

```
var objekt = {
    eigenschaft : "Objekteigenschaft"
};

function beispielFunktion () {
    // this zeigt nun auf objekt
    alert(this.eigenschaft);
}

 // Erzwinge Kontext mit apply, setze objekt als Kontext
beispielFunktion.call(objekt);
```

Der Unterschied zwischen `call` bzw. `apply` ist der dritte Parameter. Über diesen können der aufgerufenen Funktion Parameter durchgereicht werden. Während `call` die Parameter für den Aufruf einzeln erwartet, also als zweiter, dritter, vierter und so weiter, erwartet `apply` alle Parameter in einem Array.

```
var objekt = {
    eigenschaft : 0
};

function summe (a, b, c) {
    this.eigenschaft = a + b + c;
    alert("Ergebnis: " + this.eigenschaft);
}

// call: Übergebe drei einzelne Parameter
summe.call(objekt, 1, 2, 3);
// apply: Übergebe drei Parameter in einem Array
summe.apply(objekt, [1, 2, 3]);
```

Im Beispiel werden `call` und `apply` genutzt, um die Funktion `summe` im Kontext von `objekt`auszuführen. Die Funktion nimmt drei Parameter entgegen, summiert diese und speichert sie in einer Objekteigenschaft. Der Effekt der beiden `call`- und `apply`-Aufrufe ist derselbe, `summe` bekommt drei Parameter.

`call` und `apply` alleine helfen uns zur Lösung der Kontextproblematik noch nicht weiter. Sie stellen allerdings das Kernstück der Technik dar, die im Folgenden beschrieben wird.

`bind` und `bindAsEventListener`

`bind` und `bindAsEventListener` sind zwei verbreitete Helferfunktionen, die durch das JavaScript-Framework Prototype bekannt wurden. Sie werden dem Prototyp von Funktionsobjekten (`Function.prototype`) hinzugefügt. Daraufhin besitzt eine beliebige Funktion die Methoden`funktion.bind(…)` und `funktion.bindAsEventListener(…)`.

`bind/bindAsEventListene`
n funktionale Programmierung
Closures und `call/apply`, um
gewünschte Funktion im angeg
Kontext aufzurufen.

`bind` und `bindAsEventListener` erzeugen dynamisch eine neue Funktion, die die ursprüngliche Funktion umhüllt und an dessen Stelle verwendet wird. Man spricht von**Wrapper-Funktionen**.

Der Sinn dieser Kapselung ist in erster Linie die Korrektur des `this`-Kontextes. Dazu werden die besagten `call` und `apply` verwendet. `bind` und `bindAsEventListener` nehmen – genauso wie`call`/`apply` – als ersten Parameter das Objekt an, das als Kontext verwendet wird.

`bind` ermöglicht es zudem, der ursprünglichen Funktion Parameter zu übergeben, sodass darin nicht nur ein Objekt über `this`, sondern viele weitere Objekte verfügbar sind. Das Erzeugen einer neuen Wrapper-Funktion, die eine andere mit vordefinierten Parametern aufruft, nennt sich **Currying**.

Kommentierter Code

Die Funktionen `bind` und `bindAsEventListener` sehen kommentiert so aus:

`nction.prototype.bind = function () {`	Erweitere alle Funktionsobjekte um die Methode `bind` über den Prototyp aller Funktionsobjekte.
`  var originalFunction = this;`	Speichere die gegenwärtige Funktion in einer Variablen, damit in der Closure ein Zugriff darauf möglich ist.
`  var args = Array.prototype.slice.call(arguments);`	`bind` nimmt eine beliebige Anzahl von Parametern entgegen. Sie sind nicht in der Parameterliste aufgeführt, der es wird `arguments` zum Zugriff darauf verwendet. Diese Liste wird zunächst in einen echten Array umgewandel indem eine Array-Methode darauf angewendet wird.
`  var contextObject = args.shift();`	Entnehme dem Array den ersten Parameter. Das ist das Objekt, in dessen Kontext die Funktion ausgeführt werde soll. `args` enthält nun die restlichen Parameter.
`  var wrapperFunction = function () {`	Erzeuge eine verschachtelte Funktion, die als Closure wirkt. Die Closure schließt`originalFunction`, `args` und `contextObject` ein.
`      return originalFunction.apply(contextObject, args);`	Innerhalb der erzeugten Funktion: Rufe die ursprüngliche Funktion im Kontext des Objektes auf, reiche dabei die restlichen Parameter durch und gib den Rückgabewert der Funktion zurück.
`  };`	
`  return wrapperFunction;`	Gib die soeben erzeugte Wrapper-Funktion zurück.

`nction.prototype.bindAsEventListener = function (contextObject) {`	Erweitere alle Funktionsobjekte um die Methode`bindAsEventListener` über den Prototyp aller Funktionsobjekte. Die Funktion nimmt nur einen Parameter entgegen: Das Objekt, in dessen Kontext die Funktion ausgeführt werden soll.
`  var originalFunction = this;`	Speichere die gegenwärtige Funktion in einer Variablen, damit in der Closure ein Zugriff darauf möglich ist.
`  var wrapperFunction = function (event) {`	Erzeuge eine verschachtelte Funktion, die als Closure wirkt. Die Closure schließt `contextObject` und `originalFunction` ein.
`      var eventObject = event || window.event;`	Vereinheitliche den Zugriff auf das Event-Objekt. Dieses wird der Handler-Funktion entweder als Parameter übergeben (hier `event`) oder steht im Internet Explorer unter `window.event` zur Verfügung.
`      return originalFunction.call(contextObject, eventObject);`	Rufe die ursprüngliche Funktion im Kontext des Objektes auf, reiche dabei das Event-Objekt durch und g den Rückgabewert der Funktion zurück.
`  };`	
`  return wrapperFunction;`	Gib die soeben erzeugte Wrapper-Funktion zurück.

Kurzschreibweise

Ohne Kommentare und Variablen, die bloß der Lesbarkeit dienen, sehen die beiden Funktionen wie folgt aus:

```
Function.prototype.bind = function () {
    var method = this, args = Array.prototype.slice.call(arguments), object = args.shift();
```

```
    return function () {
        return method.apply(object, args);
    };
};

Function.prototype.bindAsEventListener = function (object) {
    var method = this;
    return function (event) {
        return method.call(object, event || window.event);
    }
};
```

Anwendung bei einfachen Modulen

`bind` ist die allgemeinere Funktion, die z.B. bei Timeouts, Intervallen und Callbacks Verwendung findet. `bindAsEventListener` ist die Nutzung einer Funktion als Event-Handler zugeschnitten.

Der folgenden Code zeigt, wie sich die obigen Beispiele mithilfe von `bind` und `bindAsEventListener` lösen lassen, sodass `this` immer auf das richtige Objekt zeigt.

```
var Modul = {
    eigenschaft : "Eigenschaftswert",
    start : function () {
        alert("start wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
        setTimeout(this.verzögert.bind(this), 100);
        document.getElementById("button").onclick = this.handler.bindAsEventListener(this);
    },
    verzögert : function () {
        alert("verzögert wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
    },
    handler : function (e) {
        alert("handler wurde aufgerufen\n" +
            "Event-Objekt: " + e + "\n",
            "this.eigenschaft: " + this.eigenschaft);
    },
};
Modul.start();
```

Anwendung bei Prototypen und Instanzmethoden

```
function Konstruktor () {}
Konstruktor.prototype = {
    eigenschaft : "Eigenschaftswert",
    start : function () {
        alert("start wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
        setTimeout(this.verzögert.bind(this), 100);
        document.getElementById("button").onclick = this.handler.bindAsEventListener(this);
    },
    verzögert : function () {
        alert("verzögert wurde aufgerufen\n" +
            "this.eigenschaft: " + this.eigenschaft);
    },
    handler : function (e) {
        alert("handler wurde aufgerufen\n" +
            "Event-Objekt: " + e + "\n",
```

```
            "this.eigenschaft: " + this.eigenschaft);
    };
};
var instanz = new Konstruktor();
instanz.start();
```

Hier mag zunächst die Schreibweise `this.verzögert.bind(this)` und`this.handler.bindAsEventHandler(this)` irritieren. Diese Aufrufe hüllen `verzögert` und`handler` in Closures, welche die beiden Methoden im Kontext der Instanz ausführen.

Function Binding und Currying mit solchen Helferfunktionen erlaubt das präzise Anlegen von Closures und ist ein wichtiges Werkzeug, um Verfügbarkeitsprobleme zu lösen. Im Gegensatz zu einer großen Kapselfunktion, in der alle weiteren Funktionen als Closures angelegt werden, ist Binding punktgenauer und vielseitiger – es funktioniert auch bei einfachen `Object`-Literalen und Methoden, die über den Prototyp hinzugefügt werden.

Es gibt viele, bessere Binding-Funktionen

Die hier vorgestellten `bind` und `bindAsEventListener` sind zwei *sehr einfache* Umsetzungen. Es gibt viele weitere, die mehr Komfort bieten und auf Performance optimiert sind. Die tatsächlichen Funktionen im Prototype-Framework ermöglichen es beispielsweise, dass Parameter der Wrapper-Funktion an die ursprüngliche Funktion weitergegeben werden. Die oben beschriebene `bind`-Funktion gibt lediglich die Parameter weiter, die beim `bind`-Aufruf angegeben wurden.`bindAsEventListener` aus Prototype erlaubt ebenfalls beides, während die obige Funktion nur das Event-Objekt weitergeben kann.

Nutzen Sie die ausgereiften Bi
und Currying-Funktionen aus
bekannten JavaScript-Bibliothe

Neben Prototype bieten auch andere Frameworks und funktionale Bibliotheken Function Binding und Currying. Das Konzept ist dasselbe, die Umsetzungen und Benennungen unterscheiden sich im Detail. Das Framework Mootools bietet etwa bind und bindWithEvent.

Einführung in JavaScript – Deutschsprachige Dokumentation der Programmiersprache JavaScript

Originaladresse: http://molily.de/js/

Autor: Mathias Schäfer (molily)

Kontakt: zapperlott@gmail.com